AD-A238 047

DTIC
ELECTE
JUL 10 1991
S D
C

(2)

RL-TR-91-76, Vol II (of four)
Final Technical Report
June 1991

EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES Technical Reports

Stanford University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. 5291

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

91-04336

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

91 7 8 025

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-91-76, Volume II (of four) has been reviewed and is approved for publication.

APPROVED: Northrup Fowler III

NORTHRUP FOWLER III
Project Engineer

APPROVED: Raymond P. Urtz, Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Ronald Raposo

RONALD RAPOSO
Directorate of Plans & Programs

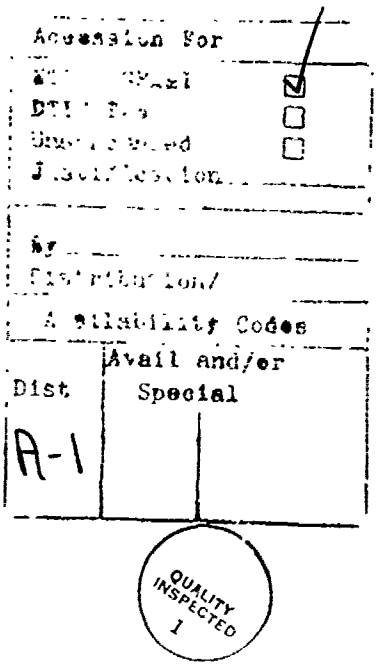
If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(COE) Griffiss AFB, NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

Best
Available
Copy

EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES, Technical Reports

Edward A. Feigenbaum
Robert Engelmore
H. Penny Nii
James P. Rice



Contractor: Stanford University
Contract Number: F30602-85-C-0012
Effective Date of Contract: 14 March 1985
Contract Expiration Date: 31 March 1990
Short Title of Work: Concurrent Expert Systems Architecture
Program Code Number: 0E20
Period of Work Covered: Mar 85 - Mar 90

Principal Investigator: Edward A. Feigenbaum
Phone: (415) 723-4878

RL Project Engineer: Northrup Fowler III
Phone: (315) 330-7794

Approved for public release; distribution unlimited.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Northrup Fowler III, RL (COE), Griffiss AFB NY 13441-5700 under Contract F30602-85-C-0012.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1991	Final Mar 85 - Oct 90

4. TITLE AND SUBTITLE
EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES, Technical Reports

5. FUNDING NUMBERS
C - F30602-85-C-0012
PE - 62301E
PR - E291
TA - 00
WU - 01

6. AUTHOR(S)
Edward A. Feigenbaum, Robert Engelmore, H. Penny Nii and James P. Rice

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Knowledge Systems Laboratory
Stanford University
701 Welch Rd, Bldg C
Palo Alto CA 94304

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
1400 Wilson Blvd
Arlington VA 22209-2308

Rome Laboratory (COE)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-91-76, II (of four)

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Northrup Fowler III/COE/(315) 330-7794

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
This final report documents the results of a five-year investigation of methods for achieving higher performance for knowledge-based systems through the design of innovative software and hardware systems architectures. Volume I summarizes the work performed and lessons learned, and serves as an annotated index to the set of over 50 project technical reports. Volumes II through IV contain the project technical reports.

NOTE: Rome Laboratory/RL (formerly Rome Air Development Center/RADC)

14. SUBJECT TERMS
Multiprocessor Architectures, Artificial Intelligence, Blackboard Systems

15. NUMBER OF PAGES
478

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500
Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Table of Contents for Volume 2

[Aiello 86]	User-Directed Control of Parallelism. The CAGE System	2-1
[Aiello 88]	Cage: The Performance of a Concurrent Blackboard Environment.	2-15
[Aiello 89]	The Cage System User's Manual.	2-26
[Bandini 89]	An Application in Poligon.	2-58
[Brown 86]	An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures.	2-73
[Byrd 87a]	A Point-to-Point Multicast Communications Protocol.	2-116
[Byrd 87b]	Considerations for Multiprocessor Topologies.	2-148
[Byrd 87c]	A Dynamic, Cut-Through Communications Protocol with Multicast.	2-155
[Byrd 88a]	A Performance Comparison of Shared Variables vs. Message Passing	2-181
[Byrd 88b]	Multicast Communication in Multiprocessor Systems.	2-196
[Byrd 89]	Support for Fine-Grained Message Passing in Shared Memory Multiprocessors.	2-205
[Davies 86]	CAREL: A Visible Distributed Lisp	2-226
[Delagi 86a]	LAMINA: CARE Applications Interface.	2-243
[Delagi 86b]	An Instrumented Architectural Simulation System.	2-272
[Delagi 87]	Instrumented Architectural Simulation.	2-294
[Delagi 88a]	CARE User's Manual.	2-301
[Delagi 88b]	ELINT in LAMINA: Application of a Concurrent Object Language.	2-446
[Delaney 86]	Multi-System Report Integration Using Blackboards.	2-459

Knowledge Systems Laboratory
Report No. KSL 86-31

April 1986

User-Directed Control of Parallelism; The CAGE SYstem

by

Nelleke Aiello

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

To Appear in the Proceedings of April 1986
DARPA conference.

Abstract

CAGE provides a framework for building and executing application programs as concurrent blackboard systems. The user controls which constructs of the blackboard system are executed in parallel.

1. Introduction

CAGE[1], Concurrent AGE[2], provides a framework for building and executing application programs as a concurrent blackboard system. With CAGE, the user can control which parts of the blackboard system are executed in parallel. A blackboard application can be implemented and debugged serially on CAGE. Once the serial version is debugged, concurrency can be introduced to different parts of the system, allowing the user to experiment with various configurations. We believe this incremental approach will facilitate the construction of concurrent problem solving systems and will teach us much about programming in a parallel environment. This paper describes the design of the CAGE system and gives detailed instructions for implementing an application, using the CAGE language and compiler [Rice 86]. We have included advice, warnings, and caveats based on our experience using CAGE.

The target parallel system architecture for the CAGE system is currently the same as that of QLAMBDA, a queue-based multi-processing Lisp ([Gabriel 84]and McCarthy) on which the parallel simulation is based. We are assuming a shared memory and a large number of processors. The user can specify his CAGE application in an extension of the L100 language, called the CAGE language, and use the CAGE compiler to generate CAGE code. CAGE runs on LOQS, a functional simulator for QLAMBDA. CAGE is implemented in ZETALISP for Symbolics 3600 machines and TI Explorers.

2. Overview of CAGE Design

CAGE is a blackboard framework system. In addition to the basic AGE [Nii 79] functionality, CAGE allows user-directed control over the concurrent execution of many of its contructs. The basic components of a system built using CAGE are:

1. A global data base (the blackboard) in which emerging solutions are posted. The elements on the blackboard are organized into levels and represented as a set of attribute-value pairs (a frame).
2. Globally accessible lists on which control information is posted (e.g. lists of events, expectations, etc.).
3. An indefinite number of knowledge sources, each consisting of an indefinite number of production rules.
4. Various kinds of control information that determine (a) which blackboard element is to be the focus of attention and (b) which knowledge source is to be used at any given point in the problem solving process.
5. Declarations that specify what components (knowledge sources, rules, condition and action parts of rules) are to be executed in parallel, and when to force synchronization. During the execution of the user's application CAGE will run these specified components in parallel.

Using the concurrency control specifications, the user can alter the simple, serial control loop of CAGE by introducing concurrent actions. CAGE allows parallelism ranging from concurrently executing knowledge sources all the way down to concurrent actions on the right- or left-hand-sides of the rules. The serial execution and parallel executions possible in CAGE are summarized below.

in KS Control
 serial: pick one event and execute associated KSs

[1]This research is supported by DARPA/RADC under contract number F30602-85-C-0012, by NASA under contract number NCC 2-220, and by Boeing Computer Services under contract number W-266875.

[2]CAGE is based on the AGE System and we have assumed here that the reader is familiar with the AGE system.

parallel:

1. as each event is generated execute associated KSs in parallel[3]
2. wait until several events are generated then select a subset and execute relevant KSs for all subset events in parallel

in KS

serial: 1. evaluate bindings
2. evaluate LHS then execute RHS of one rule whose LHS matches (in written order)
3. evaluate all LHS then execute all RHS whose LHSs match

parallel:

1. evaluate bindings*
2. evaluate all LHSs in parallel
 a. then synchronize (i.e. wait for all LHS evaluations to complete) and choose one RHS(pick one in order)
 b. then synchronize and execute the RHSs serially (in written order)
 c. execute RHS as LHS matches*

in Rule

serial: evaluate each clause then execute each action

parallel:

evaluate clauses in parallel then execute actions in parallel*
(first nil clause --> no match; first all non-NIL clauses --> match)

in clause

serial: Lisp code

parallel: Qlambda code

For more information about the concurrent options available in the CAGE System and how to specify them refer to Section IV of this paper.

3. Building applications in CAGE

In each of the following sections we will outline the application data that must be supplied by the user and how that information should be structured for use by the CAGE System. The CAGE System provides a CAGE language with which the user can write his application. The type of user-supplied information is similar to that required for applications constructed in the original AGE system. However, the structure of the user information is somewhat different from that of an AGE application.

[3]The starred options indicate the greatest use of concurrency

3.1. Blackboard Data Structure

There are two major components in the CAGE blackboard structure, the hypothesis *classes* (frequently called levels in hierarchical blackboard structures) and the hypothesis *nodes*. The user must specify the classes that make up his application's blackboard structure. For each class, the user must define the fields to be associated with the nodes created in that class. Nodes are created in those classes, either a priori by the user or dynamically while executing the user's rules. The following example shows the definition of several classes and their fields in the CAGE language.

```
Class Definitions for Model "example" :

Class name-of-levela :
    attribute1
    attribute2
    attribute3
    ...

Class name-of-levelb :
    attribute4
    attribute5
    ...
```

This will compile into two macro calls, DEFHYPOTHESIS-STRUCTURE and DEFLEVEL, which the CAGE System will in turn compile into the appropriate hypothesis structure.

```
(defhypothesis-structure
  user-hypothesis-structure
  (application-system-root)
  name-of-levela
  name-of-levelb
  name-of-levelc
              ...)

(deflevel name-of-levela
  ((attribute1 nil)
   (attribute2 nil)
   (attribute3 nil)
              ...))
```

Each of the levels(or classes) will be defined as an object with the attributes as instance variables and with the nodes as instances of those objects as they are created. (The user can define methods for the level objects which are generally used for printing information contained in the nodes on those levels.)

Definitions:

user-hypothesis-structure: A name the user gives the application's blackboard structure.

application-system-root: A handle on the above hypothesis structure for user access, generally a node where the input data, or a massaged version of the input data will reside, or the top level of a hierarchical hypothesis structure.

name-of-level: Each level or class must have a user supplied name.

node: An instance of a level, created either before or during the execution of the application, inheriting all the attributes of that level, but no values.

attribute: For each level the user must specify the names of the slots, which will become a template for the instance nodes, which in turn will contain the values used by the KSs. These values are initially NIL.

link: The user may also define links for connecting nodes. These links are defined

in the knowledge sources which use them and consist of a link name and an optional, opposite link. The value of a link on a node is the name of another node.

value: The value of an attribute depends on what was stored there by the rules and its structure depends on how it was stored. Values can be modified only by the user's initialization function and by the application rules. The structure of the values is arbitrary. How values are added or changed is explained in the knowledge source section.

3.2. Control Structure

All CAGE control information is referenced through the Control-Structure object. The major components of the Control-Structure are:

User-Initialization: This is a user-defined function, handling any initialization needed for the user's program, e.g. setting-up the appropriate blackboard structure (on top of the predefined hypothesis framework) from the input data.

Termination-Condition: Another user-defined function, which determines when the application should be terminated. The Termination-Condition can access the step-lists for events or expectations, perhaps checking for a significant event; or the blackboard, checking a particular node or nodes. It should return a non-nil value when the application is to be terminated.

User-Post-Processor: When the termination condition is true, a user supplied post processing function is invoked. This function can be used to print out the application's results in a readable form, or to handle any other post processing details.

Event-Info: This is a pointer to the Event-Information object which contains both the user-specified information on how events should be scheduled, and run-time data including the event list and the current focus event.

Expect-Info: Similar to the Event-Info pointer, this object keeps track of the expectations generated by the application and information specifying how those expectation should be scheduled.

Control-Rules: A list of of control rules defined by the user to determine when to execute which control step (event or expectation). The control rules are defined using the DEFCONTROL-RULE macro. Each control rule consists of a condition, an arbitrary LISP expression and a steptype, either event or expect. The following example of a control rule says that if there are any events pending on the event list (steplist of event-info is not null), then do an event next.

```
Example:

Control Rule : Crule-1
    Condition Part:
         If   : event-info⊖steplist
    Action part : event
```

LHS-Evaluator: The default function for evaluating the conditions of a rule if the knowledge source containing that rule has no left hand side evaluator over-riding this default. For most applications the CAGE provided function QAND will suffice. It is a serial or concurrent boolean AND depending on the parallel options selected by the user.

3.2.1. Event-Information

A blackboard system can be executed in several ways, the simplest being event-driven. This means that each time a rule action is executed the system records that change to the blackboard

as an event. Each event is added to a list called the *event list*. The scheduler selects an event from the event list to become the next *focus event*. The type of focus event is matched against the preconditions of the knowledge sources, and all the matching knowledge sources are activated. The rules of the activated knowledge sources are evaluated, those rules with satisfied conditions are executed and the cycle repeats until the termination is true.

To run a blackboard model with an event-driven control structure, certain control information must be supplied by the user.

selection-method: a function that determines which event to select from the event list. The user can write his own *best-first* selection method or use one of the CAGE provided functions, FIFO, LIFO, or AGENDA. If the AGENDA selection method is chosen, the user must also specify the agenda and an order.

agenda: An ordered list of event types supplied by the user. (See knowledge source specification for definition of event type.)

order: LIFO or FIFO order in which to check the agenda. There may be several different events of the same type on the event list.

collection rules: In some applications many events of the same type and the same node are generated and added to the event list. If the user specifies that type of event as a collection rule, then only one event is pursued and the others are *collected* and deleted from the event list.

3.2.2. Expect-Information

In an expectation-driven system, a rule may specify an expected result or change on the blackboard as one of the actions of that rule (called an expectation rule). When an expectation rule is executed, the expectation part of the rule is added to the *expectation list*. Later, when the control rules specify that an "expect" step should be executed, a focus is selected from the expectation list. If a change has occurred on the blackboard that satisfies the expect portion, actions associated with the expectation rule are executed.

Much of the information required to execute an expectation-driven system is similar to that of an event-driven system. The user must supply a selection-method, possibly including an agenda and order, and collection rules. Some additional information is required to execute expectation.

matcher: a function which defines how to match expectations to the blackboard. CAGE provides on default, PASSIVEMATCH, which simply evaluates the expectation portion of the expectation rule to see if its value is non-nil.

3.3. Knowledge Sources

CAGE knowledge sources are a partitioning of the application knowledge into sets of rules. Each knowledge source consists of some declarative information and a set of rules.

3.3.1. Knowledge Source Declarations

The definition of a knowledge source consists of more than just groups of rules. In order to properly interprets those rules, CAGE needs to know certain knowledge source control information, e.g.,

1. Under what circumstances should this knowledge source be invoked?
2. How should the rule conditions be evaluated.
3. what levels of the blackboard structure will be changed?
4. Which one or all of the rules whose conditions are true should be executed?
5. Are there any local variables or links to be defined for this KS?

The following features are available for the user to tailor a knowledge source to his own specifications:

:

Preconditions: A list of tokens, representing the *event types* used in rules. If the focus event has an event type that matches one of the knowledge source's preconditions, then that knowledge source is activated.

Levels: A list of pairs of blackboard levels or classes. The user must specify between which levels of his hypothesis structure a knowledge source makes inferences.

Links: If a knowledge source adds links between nodes on the blackboard, they must be defined here. The definition consists of a list of pairs of link names, a link and its inverse.

Hit Strategy: There are two main hit strategies available in CAGE, SINGLE and MULTIPLE. When a knowledge source with a single hit strategy is interpreted the rules of that KS are evaluated, in order, until one rule's condition evaluated to true. Then that rules actions are executed and no other rules are even considered. With a multiple hit strategy, the conditions of all rules of a knowledge source are evaluated and then all the actions of rules which successfully evaluated executed. In conjunction with either single or multiple hit strategies, the user can also specify ONCEONLY. This will cause a rule to be marked when its conditions are successfully evaluated. Its actions will be executed and it will never be evaluated again during that run of the application.

Definitions: A list of local definitions, available to all the rules of a knowledge source. The definitions are an efficiency feature to avoid the repeated calculation of the same value by all the rules. The structure is similar to that of LET, a list of pairs, a variable name and an expressions to be evaluated and assigned to the the variable. If the value is NIL it can be omitted.

Rule Order: A list of rule names, representing the rules of the knowledge source. This is the order in which the rules will be evaluated serially. Because the rules are actually defined as methods of the knowledge source to which they belong, each name should begin with a colon (:).

LHS Evaluator: The user can optionally specify a left hand side rule evaluation function for each knowledge source. There is also a default LHS evaluator specified for the entire application in the Control data. The evaluator specified here will override the default evaluator for this specific knowledge source. The LHS evaluator is a function which determines how the rule conditions are evaluated. CAGE provides several built-in functions which the user can select, including AND, for a simple boolean AND of the conditions and QAND for a concurrent boolean AND.

The following is an example of the definition of a knowledge source from the CRYPTO system written in the CAGE language.[4] The name of this knowledge source is "combine-weights", it has two preconditions, makes inferences from the Cryptoletter level of the hypothesis structure to the alphabet-letter level, defines a pair of bi-directional links, and uses the single-hit rule selection strategy. The combine-weights knowledge source also makes two definitions, possible-values gets the value NIL and lhs-evaluator the value QAND.

[4]The colons in the CAGE language are separators when separated by spaces from other words in the language. Colons indicate keywords when they directly precede a word.

```
Knowledge Source : combine-weights
  Preconditions : Confirmation, Contradiction
  Classes : Cryptoletter : alphabet-letter
  Links : Possible-Value-of : possible-Letters
  Rule Selection : Single

Definitions :
    possible-values ≡ nil
    lhs-evaluator ≡ qand
```

This compiles to the following CAGE macros.

```
(defknowledge-source COMBINE-WEIGHTS
  :preconditions (confirmation contradiction)
  :levels ((cryptoletter alphabet-letter))
  :links((possible-value-of possible-letters))
  :hit-strategy (single)
  :bindings ((possible-values))
  :rule-order (:letters )
  :lhs-evaluator qand)
```

3.3.2. Rules

CAGE rules consist of three major parts; definitions, conditions, and actions. Here is an example from CRYPTO in CAGE.

```
Rule :  letters {3}

Definitions :
    possible-values ≡
       possible-values(focus-node⊕
                       possible-letters)

Condition Part :
    If      : qand(focus-node is-cryptoletter,
                   possible-values)

Action Part :
    Changes :
        Change Type    :  Update
        Updated Node   :  focus-node
        Event Type     :  possible-assignment
        Updated Slots :
           possible-letters ↔ possible-values


;Combine the weights of identical possible
;values.
```

CAGE also provides a macro for defining rules called DEFRULE, to which the above will compile.

```
(defrule (combine-weights :letters)
  ((possible-values
     (possible-values
       ($value focus-node :possible-letters
               :all))))
  ((is-cryptoletter focus-node)
    possible-values )
  ((propose :EVENT-TYPE 'possible-assignment
            :CHANGE-TYPE 'supersede
            :HYPOTHESIS-ELEMENT focus-node
            :LINK-NODE nil
            :ATTRIBUTES-AND-VALUES
              '((possible-letters
                  ,possible-values))
            :SUPPORT 'combine-weights)
  ))
```

After specifying the knowledge source to which a rule should be added and the name of the rule, preceded by a colon, the user must specify the three major parts of the rule.

Definitions: The definition part of a rule is similar to a LET in structure. The local variables set here are available only to this rule, both in the condition and action parts, as well as other definitions of this rule. This is an optional component of a rule, and can be NIL.

Conditions: The second part of a rule contains the conditions. These can be one or more arbitrary LISP expressions which will be evaluated according to the left hand side evaluator as specified in the local knowledge source or at the control level. The conditions can reference both local variable definitions or variables bound at the knowledge source level. The CAGE system provides several access functions for retrieving values from the hypothesis structure, which can be used in the conditions of rules. It is important when writing the conditions of rules for a CAGE application to keep in mind the feasibility of running those clauses concurrently, i.e. keeping them independent of each other.

Actions: The action clauses make up the final part of a CAGE rule. These clauses have a very specific structure as evidenced by the preceding examples. The actions specify what changes are to be made to the hypothesis structure by a rule and how those changes should be made. The user must specify what node and attributes on the blackboard are to be changed, what the new links or values are, and how those changes are to be made (possibly deleting some old values). The user must also specify an event type, a name representing the type of change this action makes to the blackboard. If and when the event created by this action is selected as a focus event, this token will be matched against the preconditions of the knowledge sources to determine which KS to invoke next.

3.4. Initialization

There are two types of initialization which can occur at the beginning of a CAGE run. First CAGE must create the instances of all the application defined flavors which will constitute the executable form of the user's system. In addition, the user can do any other initialization he feels appropriate by defining his own initialization function, the name of which should be stored in the application's control structure. Since the major components of the application are defined as flavors, initialization can be done by defining :initialize or :after :init methods.

3.5. Input Data

The user must define two functions to handle his input data.

1. INPUT-PROCEDURE(Record, Time) : Given an input record, retrieved automatically at the correct time by CAGE, do what ever should be done with that input,e.g. add it to the blackboard.
2. TIME-OF-INPUT-RECORD(Record) : Given an input record, return the time stamp.

At the beginning of each run the user will be asked to specify an input data file by typing in the file name or selecting a file from a menu of pre-specified input data file names. The data file consists of records that can be read by the above two functions. A time stamp is mandatory on each input record.

4. Specifying Concurrency

CAGE supports the concurrent evaluation of pieces of knowledge. Once an application has been debugged in serial mode, the user can specify one or several knowledge source components to be executed in parallel. For example, the user might specify that the rules of the knowledge source be evaluated concurrently, or perhaps just the actions of the rules or a combination of the available options. With a minimum amount of recompilation, the user can change his parallel specifications and experiment with many different configurations.

In general more speed-up should occur as more components are run in parallel. But for some applications the overhead of setting up the new processes and inter-process communication costs will be greater than the speed-up gained by executing particular components concurrently. For example, if most or all of the knowledge sources of an application contain only one rule, then it would not be efficient to evaluate rules in parallel since for any one KS invocation there would only be one item to evaluate.

4.1. Concurrent Components

The use of knowledge sources to partition the knowledge in blackboard systems and, in particular, the structure of the knowledge sources in CAGE provide several obvious places for concurrency. The knowledge sources group the domain knowledge into independent modules, which theoretically, could be invoked independently and concurrently. Within each knowledge source the rules provide another source of parallelism, and within each rule, the clauses of the condition and action parts provide yet another. Of course not all clauses, rules or even knowledge sources are actually implemented totally independently of each other and some serialization may be necessary to correctly solve the application problem.

The following are the options for parallelism available in CAGE, grouped according to their allowed use in combination.

Clause level: can be used in combination with each other or any other parallel option.

actions: Execute the RHS action clauses of a rule in parallel. Note: When running RHS actions concurrently a non-deterministic system may result if both destructive (Supersede in CAGE) and constructive (Modify) actions occur to the same object in parallel. (Same object and attribute) A QLOOP macro is used to initiate the parallelism for loop actions, requiring recompilation of the rules containing loop actions.

lhs: Evaluate the LHS condition clauses of a rule in parallel. Note: Use the rule bindings to set any local variables tested here, insuring that the lhs clauses will be independent. A QAND macro is provided as the LHS-evaluator to initiate the concurrency for the conditions, requiring recompilation when this option is used.

rule-bindings: Evaluate the definitions of a rule in parallel. Again, these definitions should be independent of each other if their concurrent evaluation is to result in an actual speed-up.

Rule level: bindings can be used in combination with any of the other options, but only one of the rule options, single, multiple, sync or nosync can be used at a time.

bindings: Concurrently evaluate the definitions at the beginning of a knowledge source.

rules-single: Evaluate all of the conditions of the rules of a knowledge source concurrently, but only execute the actions of one successfully evaluated rule.

rules-multiple: Evaluate all of the conditions of the rules of a knowledge source concurrently, then serially execute the actions of all the successfully evaluated rules.

rules-sync: Evaluate all of the conditions of the rules of a knowledge source concurrently, then concurrently execute the actions of all applicable rules.

rules-nosync: Begin evaluating the conditions of the rules of a knowledge source in parallel and execute the actions of each rules as soon as the conditions are known to be true. With this option there is no synchronization between the left and right hand sides of rules.

Knowledge source level: Only one of the knowledge source options can be set at any one time.

kss: Invoke all the applicable knowledge sources concurrently at step selection, synchronizing by waiting for all knowledge sources to complete execution and add events to the event list before concurrently invoking a new set of kss.

kss-nosync: Invoke all applicable knowledge sources as soon as a new event is created. This option provides the least control of all the options available and does no synchronization. Many applications will have to be changed slightly to execute reasonably under these conditions, particularly removing any possible circular knowledge source invocations. To implement the parallel execution of knowledge sources without any synchronization, the control loop of CAGE was drastically altered from that described at the beginning of this paper. (See CAGE Overview.) Without any synchronization, as soon as an event is created it immediately allows all relevant knowledge sources to be invoked. No events are added to the eventlist and no focus event is ever selected. A timed loop was added to the top level control to re-invoke the user's initial knowledge source in case the system exhausts all previous events before the termination condition is satisfied.

kss-minisync: Add an event to the event list and do minimal computation at the point of synchronization before invoking the next set of knowledge sources. The main computation done is the collection and pruning of similar events, leaving fewer events to activate subsequent KSs. The mini-sync and no-sync options are different from the parallel kss option in that they don't use the serial step-selection procedure.

4.2. How to specify and change parallel components

A function, SELECT-PARALLEL-OPTIONS is provided to allow the user to quickly change the selected parallel options. SELECT-PARALLEL-OPTIONS has no arguments. A menu of parallel options will pop-up on the screen and the user can select new options or delete old ones.

5. Design Details

CAGE is currently implemented in an object-oriented style, using the Flavors feature of ZETALISP. The top level object in CAGE is called the BLACKBOARD. From the Blackboard object there are pointers to each of the principle components of the system, as follows

control-structure: all control information specified before compilation is stored here, as well as pointers to run-time control structures.

hypothesis-structure: the blackboard solution space, which must be structured by the user.

knowledge-source-list: names of the knowledge sources containing the production rules of the user's application.

user-functions: optional, user-defined functions invoked by the rules

information-structure: optional, user-defined, static data structures

A separate data structure, Parallel-Specifications, is used to store the parallel options selected by the user.

The DEFKNOWLEDGESOURCE macros will create, at compile time, an object for each knowledge source, and a set of associated methods. During the initialization process an instance of each knowledge source object is created. Other instances may be created during system execution if one of the concurrent knowledge source options is selected. One of the associated methods, SETUP-AND-START, evaluates the knowledge source definitions and initiates the rule interpretation when a knowledge source is invoked.

Each rule is created as three methods, EVALUATE-DEFINITIONS, EVALUATE-CONDITION, and EVALUATE-ACTION, associated with the rule's name using the :case method-combination feature of Flavors. The keywords of the action clause listed above are keywords in the method definitions, and therefore must be preceded by colons in the macro definition of a rule.

CAGE utilizes a global variable, PARALLEL-SPECIFICATIONS, whose value is a list of the current parallel options specified by the user. It is initially NIL and is updated using SELECT-PARALLEL-OPTIONS.

During execution CAGE prints out messages indicating the state of the execution and uses some simple graphics to help the user observe the simulation of concurrency. A set of small windows will appear on the right side of the screen, one for each process initiated by CAGE. Any state messages generated by the parallel process will appear in one of these associated windows, instead of the main terminal i/o window. There is only room to display 12 of these small i/o windows at the same time and still have them large enough and leave them up long enough to be readable. If more than 12 processes are active at the same time, the windows will overlap.

6. Future Directions

The next step for CAGE will be a reimplementation on CARE. The instrumentation in CARE will provide us with the needed tools for measuring the speed-up gained from each of the various concurrent options in the CAGE System. CAGE users will be able to implement and debug their applications in the current CAGE-on-LOQS system with its fast simulation time. Once an application is debugged it could then be run on the CAGE-CARE system for complete and accurate measurements.

References

[Gabriel 84] Gabriel, Richard P. and McCarthy, John.
Queue-based Multi-processing Lisp.
Proceedings of the ACM Symposium on Lisp and Functional programming :25 - 44, August, 1984.

[Nii 79] Nii, H. P. and N. Aiello.
AGE: A Knowledge-based Program for Building Knowledge-based Programs.
Proc. of IJCAI 6 :645 - 655, 1979.

[Rice 86] Rice, J. P.
The L100 Language and Compiler Manual.
Technical Report KSL-86-21, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

Knowledge Systems Laboratory
Report No. KSL 88-80

December 1988

Cage: The Performance of a Concurrent Blackboard Environment

by
Nelleke Aiello

Knowledge Systems Laboratory
Stanford University
701 Welch Rd. Bldg C
Palo Alto, Ca. 94304

The author gratefully acknowledges the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

Abstract

This paper describes Cage, a concurrent problem solving system which attempts to deploy parallelism within the traditional blackboard model. With Cage, the user can specify in detail which parts of his blackboard application to execute in parallel. It was hoped that by imbedding parallelism at different levels of the blackboard model a multiplicative speed-up could be achieved. The Cage system, its architecture, and its operation in a multi-processing environment are discussed. A number of experiments which were conducted to evaluate the speed-up and through-put achievable by Cage for one particular application are also presented.

1. Introduction

A common complaint about blackboard[1] systems is that they are too slow and cumbersome to be of practical use. One solution is to use multiple processors to execute different parts of the blackboard model simultaneously. Cage is a problem-solving system designed to speed-up the execution of traditional blackboard systems through parallel processing. In this paper the Cage system, its architecture, and its operation in a multi-processor environment are described. An application called Elint has been mounted on Cage and the results of experiments performed to test the speed up and throughput achieved by Cage for this application are also presented.

Work on Cage was conducted as part of the Advanced Architectures Project at the Knowledge Systems Laboratory of Stanford University, to study ways of exploiting parallelism at all different levels of a system's hierarchical structure from the application to the machine architecture level Cage uses parallelism at the problem solving level, and is further constrained to a target system architecture of shared-memory multi-processors.[2] The potential applications envisioned for this work can be characterized as performing real-time interpretations of continuous streams of errorful data, a class of applications which currently run too slowly on serial blackboard systems to be of practical use.

2. Cage System

The Cage system is an extension of the serial Age system [Nii 79]. The two systems are identical except that Cage allows parallel execution of many of its applications' components. Parallel execution in Cage can occur at different levels of granularity, based on natural divisions in the blackboard model. In this section, we will first give some background information about Age, and then we will describe Cage and how the user can specify concurrency in Cage.

2.1 Age Derivation

Age is an implementation of a serial blackboard system. It is composed of a knowledge base, in the form of *knowledge sources(KSs),* and a structured solution space, a *blackboard,* where the KSs can post interim results and read the results of

[1]It is assumed that the reader is familiar with the blackboard model and the relevant terminology. For more information, the reader is referred to [Nii 86] and [Engelmore 88].

[2]Those interested in distributed-memory architectures may want to read about Poligon, a concurrent blackboard system designed for distributed memory multi-processor machines[Rice 88] [Nii 88].

other KS executions. KSs contain *condition-action rules* that can read the blackboard and make changes on it. The blackboard is a structured set of *levels* on which objects are created and modified by the rules. These changes to the blackboard are called *events*. A scheduling mechanism, or *controller*, programmable by the user, invokes one KS at a time from among those triggered by the preceding events. Figure 1 shows the control and data flow of this serial control cycle.

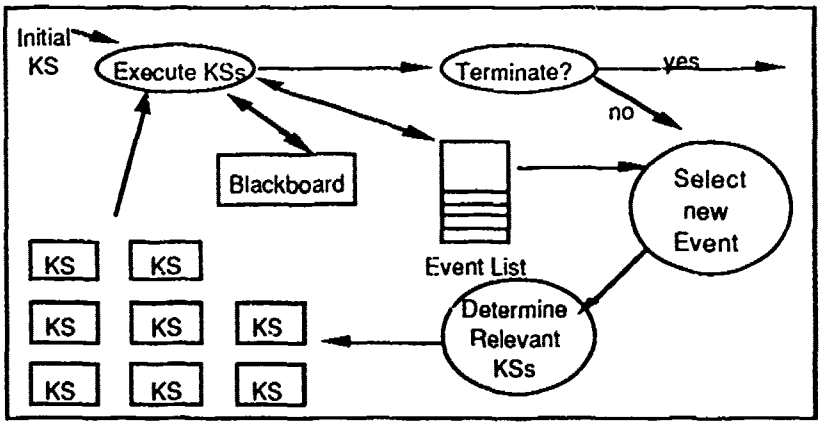


Figure 1. Serial Control Cycle

2.2 Cage Architecture

The basic components of Cage are the same as Age's with one addition--the declarations that specify which components to execute in parallel and at which points to synchronize. The components which can be executed in parallel in Cage are the KSs, the rules within the KSs, and the condition and action parts of rules. Synchronization points can be specified (1) in the control cycle between sets of concurrent KSs; (2) within a KS after evaluating all the rules' conditions but before executing any actions; or (3) within a rule, between the condition and action parts. By selecting one of the concurrency control options, the user can alter the simple, serial execution of KSs and their components so that they are executed in parallel. Next we will discuss each potential source of concurrency in detail.

Knowledge Source Concurrency

Two possible sources of concurrency exist at the KS level. A number of KSs can work either on different parts of the blackboard at the same time or in a pipeline fashion. In the application area of real-time interpretation of data, many instances of the same KS can simultaneously deal with new data items. Each of these KSs then becomes the first in a chain of KSs which interprets the data up the blackboard's levels of abstraction.

KSs in Cage can be executed in parallel with or without synchronization at the control level. With synchronization, the controller waits for all previously invoked KSs to complete before invoking the next set of triggered KSs. Without synchronization, KSs are invoked immediately when triggered, without waiting for any other KS.

Rule Concurrency

Within each KS further concurrency is possible by executing the rules in parallel. Again, Cage provides several different options for running the rules in parallel. First the condition parts of rules are evaluated. Next, if the user opts to synchronize, the controller will wait until all the conditions have been evaluated before executing the action parts of the applicable rules concurrently. The user can also specify the parallel evaluation of the conditions with the serial execution of the actions. Without synchronization, the applicable actions are executed as soon as a rule's conditions have been evaluated.

Clause Concurrency

Even finer grain concurrency is possible in Cage within each rule, by executing individual predicates of the condition part concurrently. Only one option is available, evaluation of the predicates in parallel, and execution of the action clauses in the action part of applicable rules in parallel.

2.3. Using Cage

In addition to the speed-up and through-put data about Cage gathered in the experiments described in the next section, we also learned a number of lessons about programming in a concurrent environment. Implementing the concurrency outlined above created a number of programming problems. For example, at the rule level, the state of the blackboard which lead to a rule firing may be changed before that rule's actions can be executed. Also, a rule may access values from several different blackboard objects with no guarantee that those values are consistent with each other. Memory contention can be a problem at the clause level, if a number of clauses refer to the same blackboard object at the same time, negating the benefits of concurrent execution.

Data inconsistency was alleviated by creating an atomic operation that could read and then write a blackboard object without allowing any intervening operations. In addition, a block read operation was defined, so that a rule can read all relevant information from an object with the guarantee that data will be consistent within that object. No other operations are allowed to an object during a block read of that object.

Data coherence can be maintained when running KSs in parallel, by reading all the slots of a object that are referenced in a KS at the same time, locking the object just once. This is in contrast to locking the object every time a slot is read by the rules. In other words, all necessary blackboard data is collected into local variables, called *definitions* in the KS's activation context before any rules are evaluated. Thus all the rules within a KS refer to data from the same time.

In a serial blackboard system one KS **precondition** may serve to describe several changes to the blackboard adequately. For example, suppose the firing of one rule causes three changes to be made serially. The last change, or event, is usually a sufficient precondition for the selection of the next KS. In a concurrent system, however, since those changes may occur asynchronously, all three events must be included in a KS's precondition to ensure that all three changes have actually occurred before the KS is executed. In general, a simple precondition consisting of an event token is not sufficient as it was in a serial system. A detailed specification of

the activation requirements of the KSs must be available, either in their preconditions or in the controller.

Occasionally two KSs running in parallel may attempt to change a slot at almost the same time. It is possible that the first change could invalidate the later changes. To overcome this **race condition**, a conditional action--an action which checks the value of a slot before making a change--was added.

3. Experiments

In this section we will describe seven experiments conducted with Cage,[1] the application used, and the results. The purpose of the experiments was to determine the speed-up and through-put achievable by Cage under various conditions, concurrency specifications, and resource allocation schemes. The first four experiments measured the speed-up gained by executing various blackboard components in parallel. The last three experiments related to improving the through-put of the Cage system.

The application, Elint, is a signal understanding system which integrates reports from passive radar collection sites in order to understand the positions and intentions of aircraft traveling through a monitored airspace. The application takes streams of observations from the various collection sites, abstracts them into radar emitters, tracks the emitters, groups them into clusters, and determines the intentions and degree of threat of the clusters. An emitter might be a single aircraft; a cluster could be a group of planes flying in formation or one aircraft with multiple radars systems.

Two different input data sets were used for the experiments. The first, called *Lumpy*, was a realistic data set with inconsistencies, errors and a variable number of observations per time interval. The problem with this data set was the variation in data density that made it very difficult to measure performance. Thus a second data set, *Fat*, with a constant data density was created.

Experimental Method

The method used in the experiments changed over time, based on the results of earlier experiments. In the first experiment speed-up was measured very simply, dividing the time for the application to run a given set of input data on one processor by the time for the same system executed on multiple processors. This speed-up measure did not work well, however, because the behavior of the system changed depending on how heavily or lightly it was loaded. A rate of data arrival which adequately loaded a 4 processor machine caused data starvation for 16 processors. Later experiments used a more equitable comparison scheme in which different sampling intervals were used for different numbers of processors. The sampling interval for a particular number of processors was set to be the shortest interval which still produced non-increasing latencies, where latency is the time between the input of data and the output of reports based on that data. Speed-up was measured by comparing these sampling intervals with the uni-processor sampling interval. The sampling intervals are indicators of the through-put for a particular number of processors.

[1] A more complete description of these experiments can be found in [Nii 88].

All measurements generated by the experiments were provided by the underlying Care simulator.[Delagi 86] Because Care uses a distributed memory architecture, it was necessary to emulate the shared memory model by using half the processors for processing and the other half as memory only.[1] A variation of Qlisp[Gabriel 84],a queue base Lisp including Qlet's and Qlambda's was created to program the concurrency.

3.2. Experimental Results

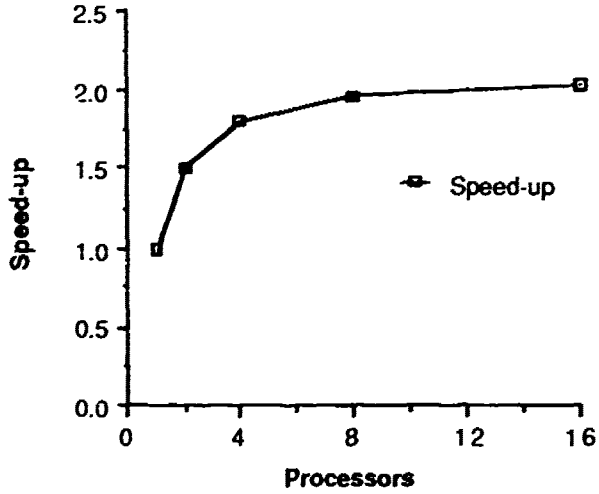


Figure 2. Results of Experiment 1

Experiment 1 measured the speed-up attainable for a varying numbers of processors with parallel KSs. For this experiment the controller started all triggered KS executions in parallel, waiting until they were done before selecting another set to run in parallel. Using the realistic "*Lumpy*" data set, this experiment exercised all the problem solving capabilities of the Elint application. Experiment 1 was run serially on one processor and then over multi-processors varying from 2 processor to 16 processors. By comparing the time required to run the data set on one processor with the time required to run with 2-16 processors, a measure of speed-up was obtained.

As show in Figure 2, the basic speed-up began to level off with 4 processors and reached 2 with 8 processors. To explain why only a factor of two speed-up was achieved, we need to look at the serial case. In the serial case (see Figure 3) the controller selects one KS to execute from among all the KSs applicable at that time.

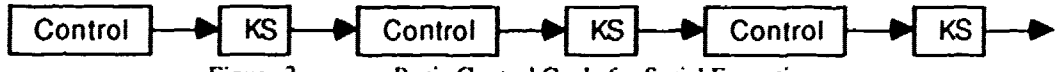


Figure 3. Basic Control Cycle for Serial Execution

In Experiment 1 all the pending KSs are executed in parallel, as seen in Figure 4.

[1]In the experimental description, the "number of processors" refers to the number of processors used for processing and does not include those used for menory only.

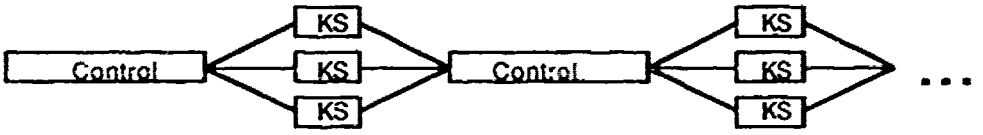


Figure 4. Basic Cycle with Serial Control and Parallel KSs

In Experiment 1 all the pending KSs are executed in parallel, as seen in Figure 4. Although the KSs were run in parallel "Amdahl's limit" limits the speed-up to the longest serial component, in this case the controller plus the longest KS. When all component parts of the Cage execution were individually timed, it was found that in the multi-processor case slightly less than half of the execution cycle time was being spent in the serial, synchronizing controller. Experiment 1 demonstrates that no matter how many KSs are run, speed-up gains are limited by the duration of the synchronizing controller and the KSs.

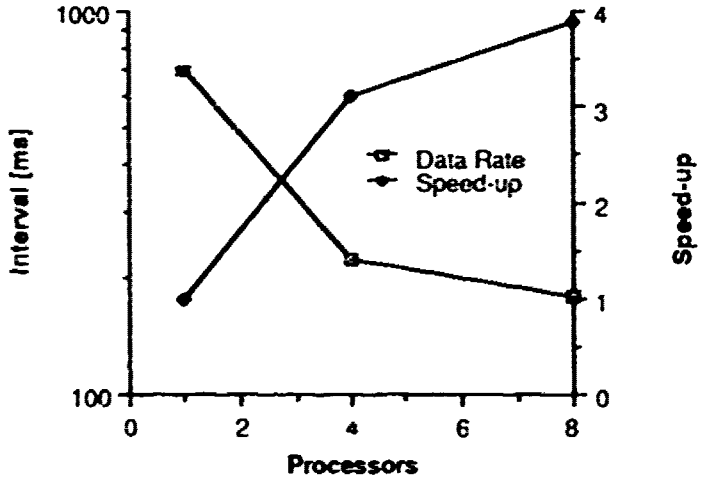


Figure 5. Results of Experiment 2

Experiment 2 also measured speed-up, but in a manner that was felt to be more *fair* than the basic speed-up experiment, using the second speed-up measure explained in section 3. This and subsequent experiments used the *Fat* data set. Experiment 2 was run with three grid sizes, 1, 4, and 8 processors. Because of what was learned in Experiment 1, in this experiment the KS were executed without synchronization, reducing the waiting time. As each KS completed, the controller immediately invoked any newly triggered KSs without waiting for any other KSs to finish.

The speed-up obtained by running KSs concurrently without synchronizing in the serial controller was slightly less than 4.(See figure 5) This is almost double the speed-up obtained with synchronization. The time spent in the controller was reduced to almost half of that in Experiment 1. But, it should be noted that the central controller is still a bottleneck. Given the architecture of blackboard systems, centralized controller time can be reduced, but not eliminated, without a major shift in the way we view blackboard systems.

Experiment 3 attempted to increase the speed-up by exploiting parallelism at a finer granularity than in Experiment 2. We hoped to gain a multiplicative increase in the overall speed-up for each KS by executing the rules in parallel. The rules were executed with both condition and action parts running concurrently and without synchronizing between the condition and action parts. Otherwise the experimental variables of Experiment 3 are identical to those of Experiment 2.

The initial results of Experiment 3 were disappointing. For 8 processors only a 5.5% speed-up over Experiment 2 was attained, for a total speed-up of 4.12. For 4 processors there was no speed-up at all over Experiment 2. The overhead of spawning processes offset any gains from more parallelism. We tried running Experiment 3 on a 16 processor grid in hopes of alleviating the congestion on the smaller grids. This resulted in slightly better results, a total speed-up of 5.6. This extra speed-up is due to the greater availability of free processors to handle the greater number of processes produced with rule level granularity.

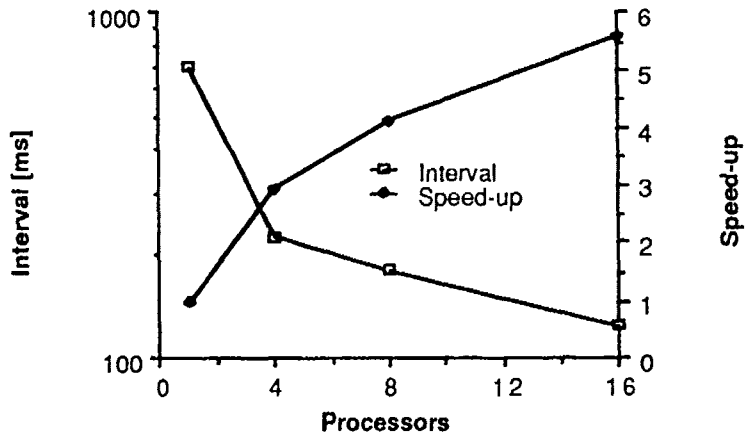


Figure 6. Results of Experiment 3

Throughout the first three experiments one troubling aspect was the apparent low sampling intervals Cage could support. (The sampling interval gives a measure of the actual through-put rate.) The minimum sampling intervals for Elint on Cage were around 120ms which was considerably slower than that of other concurrent Elint applications, such as the one done on Poligon.[Rice 88] To determine the reasons for this slow through-put various timings on all components parts of Cage were taken. As expected, most of the time was being spent setting-up and executing KSs. However, within the KSs a very high percentage of time was spent in the creation/match process; searching for existing blackboard objects or creating new ones if no match was found. A separate creation processor handles this creation/match process in Cage. A second interesting observation was that the timings were not regular, they were, in fact, very spiky. Operations that on average took only a few milliseconds occasionally took a hundred milliseconds or more. An initial hypothesis was that the spikes were caused by blocked and descheduled processes, an indication of problems in resource allocation.

Experiment 4 attempted to solve both the spikiness problem and the unexpectedly high cost of creation by allocating some of the processors to specific tasks, thus

freeing those processors from interruption by other tasks. The three most time consuming tasks were creation/match, control, and data input, so these three processes were pre-allocated to specific processors and no other processes were allow to run on those processors.

The results of this experiment were not conclusive. Experiment 4 had a speed up of 3% over experiment 3, or a total speed-up of 5.7x. But 3% falls within the margin of error for these measurements. The queue lengths for KSs and object creation/match processors increased, indicating (1)that insufficient numbers of processors were available for the KSs, because of the three pre-allocated processors and (2)that the object creation/match handler probably needed two or more processors to handle its load.

Experiment 5, a second experiment involving specialized processor allocation, was more successful. In this case only one processor, the input-handler, was used to execute the entire input procedure. Previously the creation of new input objects (observations), one for each input data item, had been handled by a separate creation handler. By eliminating the cost of spawning the separate creation process and the possibility of blocking the input process while waiting for the creation to complete, the input object creation time was decreased by 59%. Also the spikiness in the creation measurements almost disappeared.

One other improvement made in experiment 5 involved the use of a new, more accurate simulator with 4 times faster memory access. This improved the total through-put by 43%. The combination of local creation by the input handler and optimizations in the simulator improved the best sampling interval in experiment 5 from 120ms to 40ms.

Experiment	ms	% over Exp 5
Experiment 5 Single creation processor	40	n/a
Experiment 6 Multiple creation processors	31	22%
Experiment 7 Local creation	25	37%

Figure 7. Through-put Results of Experiments 5, 6, and 7

In **Experiment 6** and 7 the number of processors used was increased to 32. Preliminary runs showed little improvement in through-put due *just* to the increased number of available processors. To use those additional processors experiment 6 also increased the number of creation process handlers from 1 to 4. Separate processors were used to handle the creation of objects at different levels of the blackboard. These multiple creation handler processors together with the 16 additional processors reduced the through-put to 31ms, a 22% improvement over the best results of Experiment 5. This improvement is a strong indication that the single creation process was a bottleneck.

Experiment 7, the final experiment, was an attempt to remove the creation bottleneck completely, by doing all creation on the local processor, not on a special creation processor. In order to avoid the creation of duplicate objects, the blackboard

level object was locked by the KS until a new object was created or an existing match was found. Local creation, on the same processors as the KS or rule, also eliminated the need for Qlisp closures. Qlisp closures are one of the most expensive features of the Qlisp language which Cage uses to program in parallel, because the Qlisp closure requires the passing of the context of the local processes to the creation handler. With local creation and without the Qlisp closure through-put was improved to 25ms, or a 37% improvement over experiment 5. (See Figure 7)

4. Analysis of Speed-up and Throughput Achieved

The Cage experiments resulted in two important measurements. These are the maximum relative speed-up, comparing uni-processor runs with multi-processor runs, and the minimum sampling interval, measuring the total throughput.

4.1 Speed-up

Experiments 1 through 4 resulted in a best speed-up of 5.7x using a 16 processor grid with KSs and rules running concurrently without synchronization. The factors limiting this speed-up include:

- The existence of a central controller
- The serial definition section of KSs
- The inefficient allocation of processes to processors
- The high overhead of Qlisp closures

The affects of the central controller were minimized in experiment 2 through the elimination of synchronization at that level. The definitions, which are the local bindings done at the beginning of each KS to maintain data coherence (see section 2.3), are the only part of the KS still executed serially. Executing definitions in parallel is an option in Cage, but because of the cost of blackboard object creation (63% of the average definition time) and the difficulty in writing independent definitions, at most a 15% improvement in speed-up could be expected from concurrent definitions.

Experiments 4 and 5 showed that careful resource allocation could improve speed-up. We believe that further improvements in speed-up are possible with tailored resource allocation for additional Cage processes. While experiment 6 and 7 only measured through-put, a prelimary run under similar conditions showed a speed-up of 6. Some of this gain is also due to the elimination of the use of Qlisp closures for object creation. Qlisp closures are particularly expensive for Cage because they entail the copying of the context from the spawning processors to the executing processor. However, some Qlisp is still required to program concurrency in Cage's shared memory architecture on the underlying simulator.

4.2 Through-put

The second major result of the Cage experiments is the slow through-put achieved. The minimum sampling rate for Cage is about 9 times slower than that of a similar distributed memory system running the same application. The factors limiting speed-up also limit the through-put. In addition, it should be noted that there was no optimization of Cage or the Elint application, which could improve through-put significantly.

5. Conclusions

On the positive side, Cage can execute multiple sets of rules, in the form of KSs, concurrently. A speed-up of 4.12 was achieved by the early experiments, improved to 5.7 with optimizations of the resource allocation and 16 processors, and further improved to almost 6 with 32 processors in the last experiment. On the other hand, the use of a central controller to determine which KSs to run in parallel drastically limited speed-up, no matter how many KSs were executed in parallel. The shallow knowledge base of the application limited concurrency at the rule level, but more rules per KS would increase concurrency. Overall, we believe that, with optimization and deeper applications, Cage can be used as a viable concurrent blackboard environment.

6. References

[Aiello 86] Nelleke Aiello, *User-Directed Control of Parallelism: The Cage System.*KSL-86-31, Knowledge Systems Laboratory, CSD, Stanford Univ., April 1986.

[Delagi 86] Bruce Delagi. *CARE Users Manual.* KSL-86-36, Knowledge Systems Laboratory, CSD, Stanford University, 1986.

[Engelmore 88] Robert Engelmore and Tony Morgan (eds). *Blackboard Systems.* Addison-Wesley. Wokingham, England. 1988.

[Gabriel 84] Richard P. Gabriel, and John McCarthy. *Queue-based Multi-processing Lisp.* Proceedings of the ACM Symposium on Lisp and Functional Programming: 25-44, August, 1984

[Nii 79] H. Penny Nii and Nelleke Aiello. *AGE: A Knowledge-based Program for Building Knowledge-based Programs.* Proceedings of the 6th International Joint Conference on Artificial Intelligence: 645-655, 1979.

[Nii 88] H. Penny Nii, Nelleke Aiello and James Rice. Experiments on Cage and Poligon: Measuring the Performance of Parallel Blackboard Systems. KSL-88-66, Knowledge Systems Laboratory, CSD, Stanford University, October 1988.

[Rice 86] James Rice. *Poligon: A System for Parallel Problem Solving.* KSL-86-19, Knowledge Systems Laboratory, CSD, Stanford University, April, 1986.

Knowledge Systems Laboratory
Report No. KSL 89-86

December 1989

The CAGE System User Manual

by

Nelleke Aiello

Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA 94304

The author gratefully acknowledges the support of the following fundir g agencies for this project: DARPA/RADC, under contract F30602-85-C-0012; NASA, under contrac number NCC 2-220; and Boeing Computer Services, under contract number W-266875.

1. Introduction

This user manual describes the Cage System and its functionality. You will find a detailed description of the Cage components, the domain information the user must supply to simulate the execution an application in parallel with Cage, and the syntax for that information. We have also included listings of the user functions available in Cage and the global variables defined by Cage, as well as example domain specifications from the Elint application and a short description of the underlying simulator upon which Cage executes applications in parallel.

1.1. What is Cage?

Cage, Concurrent AGE, is an expert system shell for building and executing application programs as concurrent blackboard[Nii86] systems. With Cage, the user can control which parts of the blackboard system are executed in parallel. A blackboard application can be implemented and debugged serially in Cage. Once the serial version is debugged, concurrency can be introduced to different parts of the system, allowing the user to experiment with various configurations. We expect this incremental approach to facilitate the construction of concurrent problem-solving systems. In addition to the basic functionality found in AGE[Nii79, Aiello1981a,b], Cage allows the user direct control of the concurrent execution of many of its constructs. Otherwise, the two systems are functionally identical.

The basic components of a system built with Cage are:

1. A global data store (the blackboard) on which emerging solutions are posted. The elements on the blackboard are organized into levels and represented as a set of attribute-value pairs.

2. Globally accessible lists on which control information is posted (e.g. lists of events, expectations, etc.).

3. An indefinite number of knowledge sources, each consisting of an indefinite number of condition-action rules.

4. Various kinds of control information that determine (a) which blackboard element is to be the focus of attention and (b) which knowledge source is to be used at any given point in the problem solving process.

5. Declarations that specify which components (knowledge sources, rules, condition and/or action parts of rules) should be

executed in parallel, and when execution of components should be synchronized.

The serial control cycle begins with the selection of a knowledge source(KS) to invoke. After a change to the blackboard several knowledge sources may be applicable. Cage uses an *event list* to keep track of the changes to the blackboard and selects one of those events to match against the preconditions of the KSs. The user can specify which method to use for this event selection, such as FIFO, LIFO, or some user defined best-first mechanism. Once an event is selected, the match with the KS preconditions occurs, producing an ordered set of KSs, which are invoked one at a time. The rules of each KS are evaluated and finally the rule actions are executed for those rules with satisfied conditions. The number of rules executed depends of the KS specifications; the user may choose to allow only one rule or many rules to fire per KS invocation. Each action that is fired may cause a change to the blackboard. These changes are recorded on the event list and then the cycle repeats. Figure 1 depicts the serial Cage control cycle.

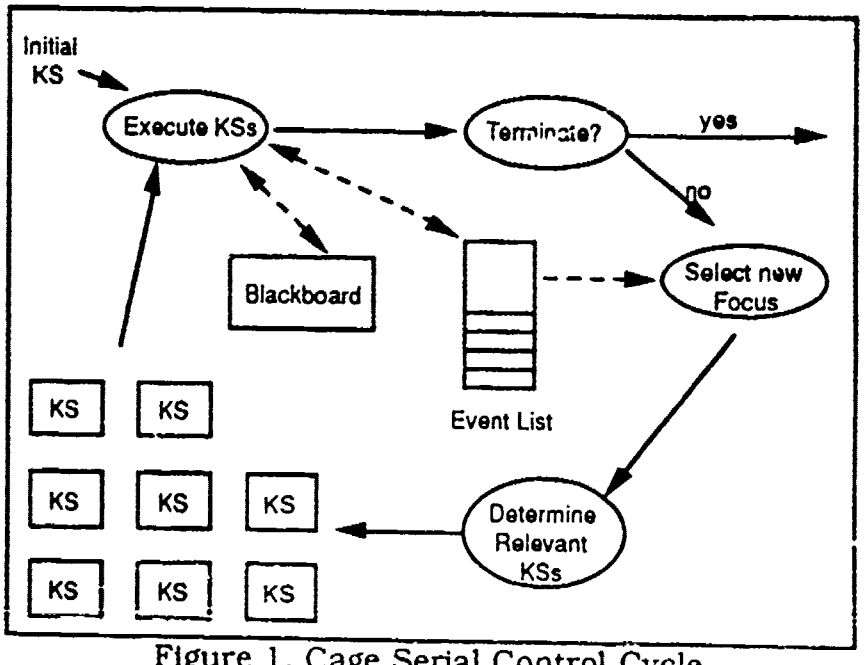


Figure 1. Cage Serial Control Cycle

Parallel control can be implemented in Cage with several variations of the serial control; by selecting more than one event from the eventlist at a time or selecting events asynchronously, thereby executing more than one KS concurrently. Cage also allows more than one rule within a KS to fire concurrently, or more than one clause within a rule, or numerous combinations of the above. The following is one possible control cycle for concurrent Cage.

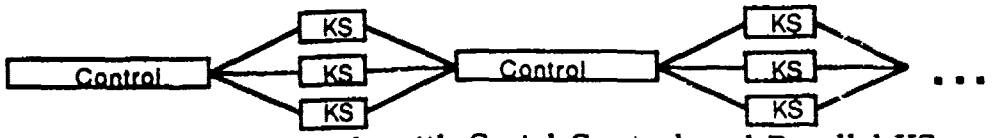


Figure 2. Basic Cycle with Serial Control and Parallel KSs

A more complete discussion of the concurrent options available in Cage can be found in Section 3.

2. Cage Application Components

Next we will describe the kind of information the knowledge engineer must supply for each of the major Cage components and how that information should be structured. Some user input is required in the specification of each of the major Cage components; the blackboard, knowledge sources, and control. The user input is similar in semantics to that required for applications constructed in the AGE system, however, the syntax is somewhat different. The concurrency specifications are outlined in a separate section of this manual.

2.1. Blackboard Structure

There are two major components in the Cage blackboard structure, the hypothesis *classes* (frequently called levels in hierarchical blackboard structures) and the hypothesis *nodes*. The user must specify the classes that make up his application's blackboard structure. For each class, the user must define the fields to be associated with the nodes created in that class. Nodes are created in those classes, either a priori by the user or dynamically while executing the user's rules. The following example from the Elint[1] application shows the definition of several classes and their fields in Cage.

```
CAGE-HYPOTHESIS-STRUCTURE

        REPORT-DATA-LEVEL
        CLUSTER-LEVEL
        EMITTER-MANAGER-LEVEL
        EMITTER-LOCATION-LEVEL
        EMITTER-LEVEL
        OBSERVATION-LEVEL


OBSERVATION-LEVEL
```

[1] All the examples in this manual are taken from Elint, a knowledge-based application which interprets real-time radar emissions from aircraft.

TIME
EMITTER-ID
SITE
LOB
OBSERVATION-TYPE
MODE
SIGNAL-QUALITY
ID-ERROR
REDIRECT-FLAG
ASSOCIATED-EMITTER

Each of the classes (or levels) will be defined as an object with the attributes as instance variables and with the nodes as instances of the class objects. (The user can define methods for the level objects which are generally used for printing information contained in the nodes on those levels.) Two macros are provided by Cage to aid the user in defining a blackboard structure, DEFHYPOTHESIS and DEFLEVEL. The following examples are again taken from the Elint application.

```
(DEFHYPOTHESIS-STRUCTURE CAGE-HYPOTHESIS-STRUCTURE
  (APPLICATION-SYSTEM-ROOT)
  REPORT-DATA-LEVEL
  CLUSTER-LEVEL
  EMITTER-MANAGER-LEVEL
  EMITTER-LOCATION-LEVEL
  EMITTER-LEVEL
  OBSERVATION-LEVEL)

(DEFLEVEL OBSERVATION
  ((TIME NIL)
   (EMITTER-ID NIL)
   (SITE NIL)
   (LOB NIL)
   (OBSERVATION-TYPE NIL)
   (MODE NIL)
   (SIGNAL-QUALITY NIL)
   (ID-ERROR NIL)
   (REDIRECT-FLAG NIL)
   (ASSOCIATED-EMITTER NIL)))
```

The DEFHYPOTHESIS-STRUCTURE function takes three arguments; the name of the hypothesis structure, a root node[1] and the node

[1] Application-system-root is provided by CAGE and should be sufficient for most applications.

names for the level in the user's particular application. The DEFLEVEL function expects the name of the level being defined and a list of pairs--the attributes and initial values (not necessarily NIL) for those attributes--which will be associated with all nodes created at the specified level.

2.2. Knowledge Sources

Cage knowledge sources are partitions of the application knowledge. Each knowledge source(KS) consists of some declarative information and a set of condition/action rules.

Knowledge Source Declarations

To interpret the rules of a KS properly, Cage needs answers to some questions about knowledge source control, for example:

1. Under what circumstances should this knowledge source be invoked?

2. How should the condition parts of rules be evaluated?

3. What levels of the blackboard structure will be changed by this knowledge source?

4. Which rule or rules out of all the rules whose condition parts are satisfied should be executed?

5. Are there any local variables to be defined for this knowledge source?

The following are the major knowledge source control options available to the user to tailor a knowledge source to his specific needs:

Preconditions: A list of tokens, representing the *event names* used in rules. If the focus event has an event name that matches one of the knowledge source's preconditions, then that knowledge source can be activated. See the Control Section for more information about events and how they are used.

Hit Strategy: The hit strategy determines how the rules are evaluated and executed; which rules have conditions which are true, how many of those rules should be fired, and in what order should the selected rules be executed. There are two main hit strategies available in Cage, Single and Multiple. When a knowledge source with a single-hit strategy is invoked, the rules of that knowledge source are evaluated, in order, until one rule's conditions are satisfied. Then the actions of the action part of the rule are executed, and no further rule is evaluated. With a multiple-hit strategy, the condition parts of all the

rules are evaluated, and all the action parts of the rules whose conditions were true are executed. In conjunction with either single- or multiple-hit strategies, the user can also specify Onceonly. This option causes a rule to be marked when its action part is executed. The marked rules will never be evaluated again during the course of a run.

Definitions: A list of local variables. The definitions are an efficiency feature to avoid the repeated calculation of the same variable. The structure is similar to that of LET pairs of variable names and expressions, except that an initial value is required for each local variable.

Rule Order: A list of rule names, representing the rules of the knowledge source. This is the order in which the rules are to be evaluated when in serial mode.

Condition-hand-side Evaluator: The user can optionally specify how the condition side of the rules within the knowledge source are to be evaluated. There is a default condition-evaluator specified in the Control data (See the Control Section). The evaluator specified within a KS will override the default evaluator for the span of that KS. The LHS-evaluator is a function which determines how the condition parts are to be evaluated. Cage provides several built-in functions which the user can select, including a boolean AND and QAND for a concurrent execution of the boolean AND, if the rule allows concurrent execution of its conditional clauses.

The macro DEFKNOWLEDGE-SOURCE is provided by Cage to aid the user in defining knowledge sources.

```
(DEFKNOWLEDGE-SOURCE <knowledge source name>
      &keywords
      :PRECONDITIONS <list of pre-condition event tokens>
      :DEFINITIONS <list of LET-type bindings>
      :KS-CONTROL<list of concurrency specifications for different
            parts of the knowledge source-->
                  (definitions <t or nil>
                   LHS <serial or parallel>
                   synchronize <t, nil or first>
                   RHS <serial or parallel>)
      :RULE-ORDER <ordered list of rule descriptors>
```

The ks-control list specifies how different components of the KS are to be executed. If definitions is followed by T then the definitions will be executed in parallel. Similarly the rule conditions (LHS) and rule actions (RHS) can be executed serially or in parallel. The value for synchronize determines whether or not to synchronize the firing of

the rules in a KS. If synchronize is T then all the rules conditions will be evaluated and then all the applicable actions. If synchronize is nil, then rule actions can fire immediately after their conditions evaluate to T. If synchronize is first, then only the actions of the first rule to evaluate to T will be executed. The following is an abbreviated example of a knowledge source specification from Elint.

```
(DEFKNOWLEDGE-SOURCE Process-Observations
      :PRECONDITIONS
            (new-observation-read)

      :DEFINITIONS
            ((the.observation FOCUS-NODE)
            ((observation-time observation-emitter-id ...)
            ($MULTIPLE-VALUES the.observation
                  ($VALUE the.observation observation-time :latest)
                        ($VALUE the.observation
                                    observation-emitter-id :latest) ...)
            ((matched-emitter-list new-emitter-node)
            ($CREATE emitter-level
                  (make-emitter
                        id  (CU:SHARED-LIST observation-emitter-id
                        emitter-type (cu:global-list  observation-type)
                        associated-observations
                              (cu:global-list the.observation)
                        last-observed
                              (CU:SHARED-LIST observation-time)
                        ...)
                  ($FIND-FOR-SLOT 'emitter-level emitter-id
                              observation-emitter-id :latest)
                  'new-or-matched-node 'process-observations-defs))
            ...)
      :KS-CONTROL (definitions nil LHS :serial synchronize :first
                        RHS :serial)
      RULE-ORDER
            (:observation-id-errorp-with-cluster
            :observation-id-errorp-no-cluster
            :inconsistent-site-observation
            :old-emitter-old-location
            :create-two-new-nodes)
      )
```

A similar macro DEFRULE is available for defining rules. The rules are evaluated according to the concurrency specifications in KS-control. A rule's condition and action clauses are evaluated according to the rule-control specifications. I.e. QAND will check the rule-control to determine whether to execute the condition clauses serially or in parallel.

```
(DEFRULE (<ks name> <:rule descriptor>)
      :IF-PART  <form>
      :ACTION-PART <form>
      :RULE-CONTROL (:lhs <t or nil>
                     :rhs <t or nil>))
```

The rule-control component specifies how the rule conditions and actions should be executed, serially or concurrently; nil or T respectively. An example illustrating the use of DEFRULE in the Elint system is given below.

```
(DEFRULE (process-observations :create-two-new-nodes)
      :IF-PART
            (QAND new-emitter-node new-emitter-location)
      :ACTION-PART
            (PROGN
                  ($SUPERSEDE new-emitter-node
                        ((emitter-my-location
                          (CU:SHARED-LIST new-emitter-location)))
                        'new-emitter
                        process-observation-2-new-nodes)
                  ($SUPERSEDE the.observation
                        ((observation-associated-emitter
                          (CU:SHARED-LIST new-emitter-node)))
                        'emitter-matched)
                        'process-observations-2-new-nodes))
                  ($MODIFY new-emitter-location
                        ((emitter-location-control-information-site
                          (LIST observation-site
                                observation-time the.observation)))
                        'new-emitter-location)
                        'process-observations-2-new-nodes)))
      :RULE-CONTROL (:lhs t :rhs nil)
      )
```

This rule has two conditions, a new emitter node and new emitter-location node have been created, and three actions, linking the two new nodes with each other and the observation node which led to their creation. In this case the concurrency specifications indicate that the conditions should be evaluated in parallel, but that the actions should be executed serially.

2.3. Control

All Cage control information is referenced through the Control-Structure object which is basically the same as in AGE. The user can specify various parts of control by setting the appropriate global variables. The major components of the Control-Structure are:

User-Initialization: This is a user-defined function, handling any initialization needed for the user's program, for example, setting-up the appropriate blackboard structure from the input data. The name of the user-initialization function should be stored in the global variable *INITIALIZER*.

Termination-Condition: Another user-defined function. This function determines when the application is to be terminated. The Termination-Condition can access the event list, expectation list, and the blackboard nodes. It should return a non-nil value when the application is to be terminated. (*TERMINATION-CONDITION*)

User-Post-Processor: When the termination condition is true, a user supplied post processing function is invoked. This function can be used to print out the application's results in a readable form, or to handle any other post processing details. (*POSTPROCESSOR*)

Event-Info: This is a pointer to the *Event-Information* object which contains both the user-specified event-scheduling information and run-time data, including the event list and the current-focus event. See the description of event-driven-control later in this section for more information on how event-information is used.

The actual structure of Event-Info is as follows:

(selection-method agenda order collection-rules matcher seeker focus steplist number-of-steps)

where *selection-method* is a function--that the user specifies in the global variable *EVENT-EXTRACTOR*--which picks an event to act upon. Items on the event *steplist* are the events that have occurred so far. Once an event is selected it is deleted from the event steplist.. The events have the following structure:

(type node hypothesis-change support rule number predecessor effect)

Agenda allows the user to specify a predetermined priority for event types and *order* specifies how to compare the events on steplist with the agenda, ie in LIFO or FILO. *Collection rules* consists of a list of event *types* which can be collapsed if more than one of that type appear on the event list. This allows the system to generate many events of the same type without forcing it to follow-up on each one individually. The *focus* is the currently selected event.[1] See the section on Event-driven control for a more detailed description of the usage of items in Event-Info.

[1] Used in serial execution.

Expect-Info: Similar to the Event-Info, this object keeps track of the *expectations* generated by the application and information specifying how those expectations are to be scheduled. *EXPECTATION-EXTRACTOR* holds the name of the *selection* funtion. Expectations also require a *matcher* function to determine when an expectation matches the current state of the blackboard.

Control-Rules: A list of control rules defined by the user to determine when to execute which control step (event or expectation). Each control rule consists of a name, a condition (an arbitrary expression), and a steptype (either event or expect). The following example of a control rule says that if there are any events pending on the event list, then do an event next.

Example:

Control Rule: Crule-1
 Condition: event-info@steplist
 Step: event

It would be defined using the specification function, DEFCONTROL-RULE, as follows:

(DEFCONTROL-RULE :Crule-1
 :condition event-info@steplist
 :step :event)

Left-hand-side-Evaluator: The default function for evaluating the condition part of rules. This default function can be over-ridden by specifying a different evaluator inside a knowledge source (see Section 2.2). For most applications the Cage-provided QAND function will suffice. It is a serial or concurrent boolean AND depending on whether the condition-side clause evaluation is to be in parallel or not. A simple boolean AND may be used, but then the clauses can not evaluated concurrently.

Input Data: The user must define a function to handle the input data, as described below. This procedure is executed by its own process, automatically inputting data according to time tags associated with the data. If the user so specifies, this process can run on a separate processor. (See Section 5.2.)

Input-Procedure(Record, Time): Given an input record consisting of a stream of time-tagged data, a record is retrieved automatically at the correct time by Cage using this function. This function should also do some actions on the data, for example, adding it to the blackboard.

At the beginning of each run the user will be asked to specify an input data file by typing in the file name or selecting a file from a menu of

pre-specified input data file names. The data file consists of records that can be read by the above function, i.e. the format of the records is left to the user. However, a time stamp is mandatory on each input record.

Event-Driven-Control: A blackboard system can be executed in several ways, the simplest being event-driven. In an event driven system each time a rule action is executed the system records that change on the blackboard as an event. Each event is added to a list called the *event list.* The scheduler selects an event from the event list to become the next *focus event.* The type of the focus event is matched against the preconditions of the knowledge sources, and all the matching knowledge sources are activated. The rules of the activated knowledge sources are then evaluated, those rules with satisfied conditions are executed and the cycle repeats until the termination condition is true.

To run a blackboard model with an event-driven control structure, the following control information must be supplied by the user.

1. A Control Rule which always returns *event* as the next step. (See Control Rule example above.)
2. Event-Info, including a *Selection-method* and possibly some *Collection rules.* (See below.) For event-driven control you do not need either a *matcher* or *seeker,* and the *focus, steplist,* and *number-of-steps* contain run-time information generated by the system.

Selection-method: a function that selects an event for focus from the event list. The user can write his own *best-first* selection method or use one of the Cage provided functions, FIFO, LIFO, or AGENDA. If the AGENDA selection method is chosen, the user must also specify the events on the agenda and their order. In this case the Selection Method should have the form (AGENDA <order><an agenda of ordered event names>).

Agenda: An ordered list of event names supplied by the user.

Order: LIFO or FIFO order in which to check the agenda. There may be several different events of the same type on the event list.

Collection rules: In some applications many events of the same type and on the same node are generated. By specifying an event name in the *collection-rules* list of *Event-Info,* only one of the events is pursued while the others are *collected* and deleted from the event list.

3. Concurrency Specifications

Using the concurrency control specifications, the user can alter the simple, serial control loop of Cage by introducing concurrent actions. Cage allows parallelism ranging from concurrently executing knowledge sources all the way down to concurrently executing the conditions and actions of the rules. The serial execution and parallel executions possible in Cage are summarized below.

Knowledge Source Control

serial:

Pick an event and execute the associated knowledge sources.

parallel:

1. As each event is generated, execute the associated knowledge sources in parallel.[1]
2. Wait until all active knowledge sources complete execution, generating a number of events, and then execute concurrently the knowledge sources relevant to those events.
3. Wait until several events are generated then select a subset and execute the relevant knowledge sources for all the subset events in parallel.

Within Knowledge Sources

serial:

1. Evaluate the bindings.
2. Evaluate the condition sides, then execute the action sides of one rule whose condition side matched.
3. Evaluate all the condition sides then execute serially all the actions of rules whose condition side matched.

parallel

1. Evaluate the bindings in parallel.*
2. Evaluate all condition sides in parallel,
 a. then synchronize (i.e. wait for all the condition side evaluations to complete) and choose one action side, or
 b. synchronize and execute the actions serially (in lexical order), or
 c. execute the actions in parallel as the condition side matches.*

Within Rules

serial:

Evaluate each clause then execute each action.

[1]The starred options indicate the greatest use of concurrency.

parallel:
Evaluate the condition-side clauses in parallel then execute actions of the action side in parallel.*
(First nil clause --> no match;
all non-NIL clauses --> match.)

Within the clauses
serial:
Lisp code

parallel:
Qlambda code

3.1. Syntax of Parallel Specifications in Cage

At the top level, the user can specify how the knowledge sources should be executed, serially or in parallel, with or without synchronization. The variable *CAGE-CONTROLS* contains a list of keywords and values including the *:knowledge-sources* to control the KS execution. The allowable values are *:no-synchronization*, *:knowledge-sources*, and *nil*, which run the knowledge sources in parallel without synchronization, with synchronization, and serially, respectively. If the KSs are run with synchronization then the control loop will wait for all the KSs to complete before invoking the next set of KSs in parallel from all the events added to the eventlist by the previous set of KSs. If the KSs are executed without synchronization then a change made by a KS is not recorded on the eventlist but instead immediately invokes any subsequent KSs.

Within a knowledge source the user can specify how the components of that KS should be executed, serially, in parallel, and with or without synchronizing. Using the Defknowledgesource function, the concurrency can be specified using the keyword *:ks-control*. The definitions of KS can be executed either serially or concurrently, as can the condition sides of the rules and the action sides. If synchronize is *T* then all of the conditions are evaluated, waiting for completion, before the actions are executed. If synchronize is *first* then Cage evaluates all the conditions in parallel, but only waits for the first rule who's condition evaluates to T before executing that rule's actions. If synchronize is *nil* then Cage executes a rules actions immediately after its conditions evaluate successfully, without waiting for any other rules.

:ks-control (definitions <t or nil>
LHS <serial or parallel>
synchronize <t, nil or first>
RHS <serial or parallel>)>

In a rule the user can control the execution of the condition and action clauses. In Defrule the keyword is *:rule-control.*

:rule-control (:lhs <t or nil>
:rhs <t or nil>)

3.2. Parallel Functionality for Cage

A number of special purpose functions and macros were added to Cage to facilitate the implementation of concurrency in a shared-memory model. These include the ability to lock nodes while accessing slot values, the locking of blackboard levels while creating or deciding to create a new node, new conditions allowed on the action side of rules, an input handler and a trace mechanism. The functions beginning with a $ sign are intended for general use by application builders, and are described in the next section with the rest of the Cage user functions.

The functions beginning with the package designation *cu:* and described below are lower level functions from the system which emulates Cage's shared memory model on the Care simulator's distributed memory. These functions provide access to data structures between processors. Local data structures can only be accessed by the local processor, they must be copied into dynamic space before other processors can access them. For more detailed information about these functions and the shared memory architecture, see [Saraiya 89] and [Delagi 87].

cu:cache-shared-structure(list)-function
List is a shared list in "dynamic space"--thus this function copies the list into "local space" corresponding to the processor at which the call is made and returns the new copy.

cu:in-memory(site &body body)-macro
Ensures that any data structures created in "dynamic space" will be in the memory module corresponding to *site*, within the dynamic scope of IN-MEMORY.

cu:shared-list(&rest elements)-function
Constructs a list in "dynamic space" with *elements*; returns it.

cu:shared-lock(lock-address)-function
Locks a spin-lock; *lock-address* is the remote-address of a memory cell in dynamic space. Returns when lock is acquired.

cu:shared-read(address)-function

Read the contents of memory cell corresponding to remote-address, *address*.[1]

cu:shared-write(address contents)- function
Write *contents* into the memory cell corresponding to remote=address, *address*.

cu:without-clock(expression)
The *expression* is executed off the clock. This macro is useful for debugging and I/O that should not be timed with simulation of the user's application.

domain-ms--global variable
Contains the data rate in milliseconds. This should be set by the user.

domain-time()--macro
Returns the domain time, ie simulation time minus the base time in domain ms.

select-parallel-options()--function
Displays a menu of parallel options, allows the user to select options, and changes the default specifications to the user's specifications.

parallelp(option)--function
Returns T if option is set to execute in parallel.

4. Cage User Facilities

In this section we describe a set of user functions, macros, and global variables provided by Cage to access the blackboard and generate events from the knowledge source rules. We would discourage the user from building rules which access the blackboard through other means, or from accessing the blackboard from outside the rules. Many of the following functions and macros have side affects, such as maintaining history lists, necessary to the proper functioning of Cage.

4.1. User Functions[2] and Macros

$modify($node accessor-value-pairs change-type &optional support)
Add a new value to the list of existing values of *$node* and attribute. *Accessor-value-pairs* is a list of pairs, of slot *(Accessor)*

[1] Automatic coding/decoding of structured data occurs on transfers between "local" and "dynamic" spaces. See pp 2-4 of [Saraiya 89].

[2] Unless otherwise stated, all the items in this section are Lisp functions.

indicating the slot to be changes and the new value to be added to the existing values of *node* and slot.

$supersede($node accessor-value-pairs change-type &optional support)--macro

Add a new value, deleting all old values of node and attribute.

$supersede-if(condition $node accessor-value-pairs change-type &optional support)--macro

If *condition* is true, make a change to the blackboard at *$NODE* and *accessors* slots, replacing all the old values with new *value* from *accessor-value-pairs*

$create(level-name creation-form finder-form change-type &optional support)--macro

Create a new node(s), using *creation form*, on the blackboard if the node specified by *finder-form* doesn't exist yet. $Create does not allow the creation of two copies of the same node by different processes because it locks the level while creating the first, and checks for existing nodes when trying to create the second. *Change-type* indicates which event is causing this creation. *Support* can be the name of the knowledge source containing the rule with this call to $Create or other documentation.

$value(node attr &optional selector)

Read the blackboard *node* and *attribute*, returning a value stored there. If *selector* equals *:latest* return the newest (ie. latest) value. If selector is not given or not equal to *:latest*, return all the values. $value is an explicit macro to allow Cage to keep track of all references to the blackboard.

$multiple-values($node &body $value-forms)--macro

Retrieve several values from the blackboard at the same time, with only one call to the *$node* and thus only one memory request to the process where that node resides. This macro allows the user to retrieve multiple values from a node without interruptions, ie. no write can change any values on the selected node until all the requested values have been read.

$find-for-predicate(collection predicate satisfaction)

Finds all nodes in *collection* that satisfy *predicate*. If *satisfaction* is non-nil, $find-for-predicate will return all nodes in *collection* that do not satisfy *predicate*

$find-for-slot(collection slot value select-type satisfaction)

Find all nodes in set *collection* which have a *slot* with value equal to *value*. *Collection* can be a level specification or a set of levels. If *satisfaction* is non-nil, $find-for-slot will return all nodes in *collection* that do not have the specified slot and value.

$propose(&key event-type change-type hypothesis-level hypothesis-element link-node attribute-and-values support comment)

Allows the user to alter the blackboard from outside a knowledge source and generates events for those changes. This function is useful in dealing with input data, ie. writing data on the blackboard or recording preprocessing results on the blackboard. $propose should not be used in general. The KSs should be the only component to write on the blackboard and generate events.

$null-event($node change-type &optional support)--macro

Generate a null event, i.e. add an event to the eventlist without making a change to the blackboard.

$LIFO

Last-In, First-Out event scheduler.

$FIFO

First-In, First-Out event scheduler.

$ALL

Select all events at the same time.

4.2. Global Variables

The following global variables are available to the user to aid in the development and execution of Cage applications.

default-parallel-specifications

Global list of components to execute in parallel, generated by the system from the user's concurrency specs.

cage-ks-names

List of knowledge sources in an application, generated as the knowledge sources are defined.

propose-history

History list of all event generating actions.

blackboard

Pointer to the top level object in the blackboard. The levels can be accessed from here.

level-names

List of names of the levels on the application blackboard.

focus

The current focus event, the latest event selected from the eventlist.

base-time
The real time that the current simulation started, used to calculate the time passed since the start of the simulation.

5. Care Components

Underlying the Cage system is a model of a MIMD, shared-memory, parallel architecture. This model and Cage have actually been implemented on two different simulators. One (Cage-Loqs) is implemented on a low-overhead version of Qlambda[Gabriel84]. The second is Cage-Care which is built on a distributed, parallel simulator called Care[Delagi86a]. This manual is intended to describe the Cage-Care implementation.

Cage-Loqs is intended for quick conversions of serial blackboard systems to parallelism. It uses the Cage language for a clear, non-lisp representation of the user's rules. It executes a simulation relatively quickly, showing the user where and when concurrency is being exploited. Using Cage-Loqs one can debug a parallel blackboard system and calculate the potential concurrency quickly. Cage-Loqs assumes there are as many processors available as needed. The major disadvantage of Cage-Loqs is that it cannot make accurate measurements of the parallelism achieved. The Cage-Loqs system is described in earlier Cage documentation[Aiello86], and the Cage language is a variation of the L100 language[Rice86b]. Cage-Care is intended for detailed simulations, measuring many factors during the simulation. It therefore executes an application much slower (1-2 orders of magnitude) than either its serial version or the Loqs version.

5.1. Shared-Memory model

In order to simulate a shared memory machine on a distributed simulator we have set up the following model. The odd numbered rows of the grid of processors are used as regular processors to run the generated processes in parallel. The even numbered rows are used only as memory. Since the grid is fully connected, the processes will have roughly equal access to all parts of the memory as in a shared memory machine.

5.2. Circuits

Currently, Cage can run on CARE with circuits of different sizes varying from 2 to 32, ie 1 to 16 processing processors and the same number of memory processors. *CIRCUIT* is a global variable through

which the user can determine which size circuit to use. The user can specify that certain processes be executed on specific processors by setting the variables *Control-Processor*, *Input-Handler-Processor*, and *Creation-Handler-Processor*. The user can also allocate different memory sites for various application data. The following is a typical Cage configuration file for a 16-site circuit [8 processors].

```
(setq *Control-Processor*                '(2  3))
(setq *Input-Handler-Processor*    '(1  1))
(setq *Creation-Handler-Processor*          '(1  3))
(setq *QLisp-Task-Queue-Memory*             '(3  2))
(setq *Blackboard-Memory*          '(1  4))
(setq *Control-Memory*             '(2  4))
(setq *Hypothesis-Memory*          '(2  2))
(setq *Input-Data-Memory*          '(1  2))
(setq *Level-Memories*                      '())
      ; application dependent alist ((level . location)...)
```

5.3. Data Rates

DOMAIN-MS is a global variable containing the rate, in milliseconds, at which data is read. Data is read into the system by an independent data handler as described earlier in the Control Section.

5.4. Instruments

A number of different instruments are available from the CARE system which allow the user to monitor the simulation of his application. The CARE User Manual[Delagi 86] describes in detail how to use these instruments.

6. How to run Cage

Cage was designed for use on a Symbolics LISP machine under release 6.1 and under the compatible version of Care, released on June 8, 1988. The top level, calling function for Care-Cage is BB:BOOT-CAGE. Assuming the user has defined all the necessary application components, BOOT-CAGE will initialize the system, start the input handler and pass control to the control loop. A sample start-up file, illustrating the load-up procedure and other essential initializations, is listed in Appendix C of this manual.

References

[Aiello 81a] Nelleke Aiello, Conrad Bock, H. Penny Nii, and William C. White, *Joy of AGE-ing: An Introduction to the AGE-1 System*. HPP-81-23, Heuristic Programming Project, CSD, Stanford University, October 1981.

[Aiello 81b] Nelleke Aiello, Conrad Bock, H. Penny Nii, and William C. White, *The AGE Reference Manual.* HPP-81-24, Heuristic Programming Project, CSD, Stanford University, October 1981.

[Aiello 86] Nelleke Aiello, *User-Directed Control of Parallelism: The Cage System.*KSL-86-31, Knowledge Systems Laboratory, CSD, Stanford University, April 1986.

[Aiello 88] Nelleke Aiello, *Cage: The Performance of a Concurrent Blackboard Environment.* KSL-88-80, Knowledge Systems Laboratory, CSD, Stanford University., December 1988.

[Delagi 86] Bruce Delagi. *CARE Users Manual.* KSL-86-36, Knowledge Systems Laboratory, CSD, Stanford University, 1986.

[Delagi 87] Bruce A. Delagi, Nakul P. Saraiya, and Greg T. Byrd. *LAMINA: CARE Applications Interface.* KSL-86-67, Knowledge Systems Laboratory, CSD, Stanford University., November 1987.

[Engelmore 88] Robert Engelmore and Tony Morgan, (eds). *Blackboard Systems.* Addison-Wesley. Wokingham, England. 1988.

[Gabriel 84] Richard P. Gabriel, and John McCarthy. *Queue-based Multi-processing Lisp.* Proceedings of the ACM Symposium on Lisp and Functional Programming: 25-44, August, 1984

[Nii 79] H. Penny Nii and Nelleke Aiello. *AGE: A Knowledge-based Program for Building Knowledge-based Programs.* Proceedings of the 6th International Joint Conference on Artificial Intelligence: 645-655, 1979.

[Nii 86] H. Penny Nii. Blackboard Systems KSL-86-18, Knowledge Systems Laboratory, CSD, Stanford University, April 1986. Also in AI Magazine, Vol. 7-2 and vol 7-3, 1986.

[Nii 88a] H. Penny Nii, Nelleke Aiello and James Rice. *Frameworks for Concurrent Problem Solving: A Report on Cage and Poligon.* KSL-88-02, Knowledge Systems Laboratory, CSD, Stanford University, March 1988.

[Nii 88b] H. Penny Nii, Nelleke Aiello and James Rice. *Experiments on Cage and Poligon: Measuring the Performance of Parallel Blackboard Systems.* KSL-88-66, Knowledge Systems Laboratory, CSD, Stanford University, October 1988.

[Rice 86] James Rice. *Poligon: A System for Parallel Problem Solving.* KSL-86-19, Knowledge Systems Laboratory, CSD, Stanford University, April, 1986.

[Rice 86b] James Rice. The Poligon User's Manual, KSL-86-10, Knowledge Systems Laboratory, CSD, Stanford University.

[Rice 88] James Rice. The Advanced Architectures Project. KSL-88-17, Knowledge Systems Laboratory, CSD, Stanford University, March, 1989.

[Rice 89] James Rice and Nelleke Aiello, *See How They Run... The Architecture and Performance of Two Concurrent Blackboard Systems*. KSL-89-08, Knowledge Systems Laboratory, CSD, Stanford University, January, 1989.

[Saraiya] Nakul P. Saraiya. *A Shared Memory Lisp Package for CARE*. KSL-88-85, Knowledge Systems Laboratory, CSD, Stanford University, January, 1989.

Appendix A. Global Variables

Values for the following global variables should be specified by the user as part of his/her application specification. Most are also described earlier in this manual in the sections that pertain to their functionality.

initializer
(FUNCTION <name of initialization function>)

termination-condition
(FUNCTION <name of termination function>)

user-post-processor
(FUNCTION <name of post processing function>)

event-extractor
A function to select events off the eventlist. LIFO and FIFO are provided or the user can write his/her own function.

expectation-extractor
A function to select events off the expectation list. LIFO and FIFO are provided or the user can write his/her own function.

Cage-Controls
A set of rules to determine what type of item to handle next, generally an event or expectation.

debugging-cage
Set to T if in debugging mode, to generate extensive traces of the application execution.

input-data-file
The file path specification for the input data of the application.

time-trace-file
The file path specification for output trace file.

domain-ms
The number of milliseconds to wait between data readings in simulated time.

Control-Processor
Specify which processor in the CARE circuit to use as the control processor , ex, '(2 3).[1]

Input-Handler-Processor
Specify which processor in the CARE circuit to use as the input handler, perhaps '(1 1).

Creation-Handler-Processor
Specify which processor in the CARE circuit to use as the creation handler, '(1 3).

QLisp-Task-Queue-Memory
Specify which processor in the CARE circuit to use for the QLisp task queue, '(3 2).[2]

Blackboard-Memory
Specify which processor in the CARE circuit to use as the blackboard memory, '(1 4).

Control-Memory
Specify which processor in the CARE circuit to use as the control memory, '(2 4).

Hypothesis-Memory
Specify which processor in the CARE circuit to use as the hypothesis memory, '(2 2).

Input-Data-Memory
Specify which processor in the CARE circuit to use as the input data memory, '(1 2).

Level-Memories
The level memories are assigned dynamically by Cage, depending on the application. The structure of this variable is an alist, associating the level with its processor, ((level . location)...), initially '().

[1]Remember, processors in the odd numbered rows of the processor grid are used as processors.

[2]Even numbered processors are used for memory in this shared memory model.

Appendix B. Cage Macros and Functions

The following functions and macros are available to the user to develop a Cage application. Many of them are also described earlier in this manual in their respective, relevant sections. For examples of their use, see the sample application appendix at the end of this manual.

Defhypothesis-structure(<name of blackboard> (<name of root>)
. <list of levels>)

Generates a hierarchical hypothesis structure of the given levels for a blackboard.

Deflevel(<level name> . <list of slot names and initial value pairs>)

Defines a level of the hypothesis structure with given slots and initial values, which can be nil.

Defknowledgesource(<knowledge source name>
:preconditions <list of preconditions event tokens>
:definitions <list of let-type bindings>
:KS-control <list of concurrency specs>
:rule-order <ordered list of rule descriptors>)

Defines a knowledge source with the given specifications.

Defrule((<ks name> <:rule descriptor>) :if-part <form>
:action-part <form>
:rule-control <rule concurrency specs>)

Defines a rule with the given specs within the ks named.

Defcontrol-rule(<name> :condition <s-expression>
:step <EVENT/EXPECTATION>)

Define a control rule to invoke a given step.

Appendix C. Sample Cage start-up file

```
;;;-*- Mode: Lisp; Package: User; Base: 10; Syntax: Zetalisp -*-

(SEND TV:SELECTED-WINDOW :SET-MORE-P NIL)
(SETF SI:PRINLEVEL 3)

;Either load the system files for the following systems or add them to
your local system directory.
(MAKE-SYSTEM "CARE" :SILENT :NOWARN :NOCONFIRM)
(MAKE-SYSTEM "QL" :SILENT :NOWARN :NOCONFIRM)
(MAKE-SYSTEM "CC" :SILENT :NOWARN :NOCONFIRM)
```

```
(MAKE-SYSTEM "ELINT-CARE" :SILENT :NOWARN
        :NOCONFIRM)

(GC-OFF)
;Load a CARE circuit, i.e octorus-16
(CU:SIMPLE :design 'OCTORUS-16 :RUN NIL)

;Assign processes to processors.
(setq *Control-Processor*            '(2  3))
(setq *Input-Handler-Processor*      '(1  1))
(setq *Creation-Handler-Processor*       '(1  3))
(setq *QLisp-Task-Queue-Memory*          '(3  2))
(setq *Black' oard-Memory*           '(1  4))
(setq *Control-Memory*               '(2  4))
(setq *Hypothesis-Memory*            '(2  2))
(setq *Input-Data-Memory*            '(1  2))
(setq  *Leve'-Memories*                    '())

;Care specs for Cage, simulating shared memory instead of distributed.
(SETQ CARE:***PRODUCTION-CARE-RUN*** T)
(PUTPROP :VALUES-MEMORY-REQ T 'C:EVALUATOR-WAIT)

;specify the input and output files for this run.
(SETQ BB:*INPUT-DATA-FILE* "device:>yourdirectory>all-at-
once.obs)
(SETQ BB:*OUTPUT-TRACE-FILE* "device:>yourdirectory>any name
                                                    you like")
;Set desired control specs and debugging flags.
(SETQ BB:*CAGE-CONTROLS* '(:NO-SYNCHRONIZATION
                                :KNOWLEDGE-SOURCES))
(SETQ BB:*DEBUGGING-CAGE* T)
(SETQ QL:***DEBUGGING-QL*** NIL)
(SETQ CU:***COUNT-LOCKS*** NIL)
(SETQ BB:***TIME-trace*** NIL)
(SETQ BB:*DOMAIN-MS* 50.0)     ;controls the data rate

(SETQ SI:GC-RECLAIM-IMMEDIATELY-IF-NECESSARY T)

;Load the patch files.
(SETF *CAGE-PATCHES*
      '(
        'device:>yourdirectory>ql>Stop-Process-Patch"
        "device:>yourdirectory>ql>memory-request-patch"
        'device:>yourdirectory>ql>Areas"
        "device:>yourdirectory>Symbolics-GC"
        'device:>yourdirectory>time-trace"
        "device:>yourdirectory>no-creator-processors
       )
      )
```

```
(MAPC 'LOAD *CAGE-PATCHES*)

(DEFUN BB:RUN ()
       (UNWIND-PROTECT
              (GC-ON :dynamic t :ephemeral nil)
              (BB:BOOT-CAGE)
              )
       (B:KILL-SIMPLE-HARDCOPY) ;don't tie-up the printer after
                                ;long runs.
       (CLOSE BB:*INPUT-DATA-STREAM*)
       )
  )

;Call BB:RUN to start the simulation.
```

Appendix D. Sample Cage application specifications

D.1. Hypothesis Structure

```
(DEFHYPOTHESIS-STRUCTURE CAGE-HYPOTHESIS-STRUCTURE
   (APPLICATION-SYSTEM-ROOT)
   REPORT-DATA-LEVEL
   CLUSTER-LEVEL
    EMITTER-MANAGER-LEVEL
    EMITTER-LOCATION-LEVEL
    EMITTER-LEVEL
   OBSERVATION-LEVEL
  )
(DEFLEVEL REPORT-DATA
        ((NEW-CLUSTERS NIL)
        (CLUSTER-SPLITS-AND-MERGES NIL)
        (CLUSTER-REFINEMENTS NIL)
        (CLUSTER-THREATS NIL)
        (REPORT-SITUATIONS NIL)
        (EMITTER-THREATS NIL)
        (NUMBER-OF-CLUSTERS NIL)
        (CLUSTER-NODES NIL)
        (ACTIVITY NIL))
  )
(DEFLEVEL CLUSTER
        ((ID NIL)
        (LAST-UPDATE NIL)
        (FIX-HISTORY NIL)
        (SPEED-HISTORY NIL)
        (HEADING-HISTORY NIL)
        (PLATFORMS NIL)
        (NUMBER-OF-PLATFORMS NIL)
```

```
      (ACTIVITY NIL)
      (SPLITS-AND-MERGES NIL)
      (ASSOCIATED-EMITTERS NIL))
 )
(DEFLEVEL EMITTER-MANAGER
      ((RECYCLE-CANDIDATES NIL)
      (EMPTY-NODES NIL)
      (NODES NIL))
 )
(DEFLEVEL EMITTER-LOCATION
     ((LOB NIL)
      (FIX NIL)
      (HEADING NIL)
      (ID NIL)
      (LAST-OBSERVED NIL)
      (MY-EMITTER NIL )
      (CONTROL-INFORMATION-SITE NIL)
      (CLUSTER-SPLIT-DEMON NIL))
 )
(DEFLEVEL EMITTER
      ((ID NIL )
      (EMITTER-TYPE NIL )
      (STATUS NIL )
      (CONFIDENCE NIL )
      (ID-ERROR NIL)
      (LAST-OBSERVED NIL )
      (NO-OF-OBSERVATIONS NIL)
      (ASSOCIATED-OBSERVATIONS NIL )
      (ASSOCIATED-CLUSTER NIL )
      (OBSERVATIONS-RECYCLED  (cu:shared-list NO))
      (CLUSTER-RECYCLED  (cu:shared-list NO))
      (MY-LOCATION NIL ) (CLUSTER-DEMON NIL)
      (THREAT-CHECK-DEMON (cu:shared-list YES))
      (RECYCLE-CANDIDATES NIL)
      (EMPTY-NODES NIL))
 )
(DEFLEVEL OBSERVATION
      ((TIME NIL )
      (EMITTER-ID NIL )
      (SITE NIL )
       (LOB NIL )
      (OBSERVATION-TYPE NIL )
      (MODE NIL )
      (SIGNAL-QUALITY NIL )
      (ID-ERROR NIL )
       (REDIRECT-FLAG NIL )
      (ASSOCIATED-EMITTER NIL ))
 )
```

D.2. Knowledge Base

```
(DEFKNOWLEDGE-SOURCE Process-Observations
  :PRECONDITIONS
  (new-observation-read)

  :DEFINITIONS
  ((the.observation FOCUS-NODE)

   ((observation-time observation-emitter-id ...)

    ($MULTIPLE-VALUES the.observation
         ($VALUE the.observation observation-time :latest)
         ($VALUE the.observation observation-emitter-id
                      :latest)
                 ...)
    ((matched-emitter-list new-emitter-node)
     ($CREATE emitter-level
              (MAKE_EMITTER
                 id  (CU:SHARED-LIST observation-emitter-id
                 emitter-type (CU:SHARED-LIST  observation-type)
                 associated-observations
                           (CU:SHARED-LIST the.observation)
                 last-observed  (CU:SHARED-LIST observation-time)
                           ...)
              ($FIND-FOR-SLOT 'emitter-level emitter-id
                           observation-emitter-id
                           :latest)
              'new-or-matched-node 'process-observations-defs))
   ...)

  :KS-CONTROL (definitions nil lhs :serial synchronize :first
                      rhs :serial)
 :RULE-ORDER
   (:observation-id-errorp-with-cluster
     :observation-id-errorp-no-cluster
     :inconsistent-site-observation
     :old-emitter-old-location
     :create-two-new-nodes)
  )
 (DEFRULE (process-observations :create-two-new-nodes)

   :IF-PART
   (QAND new-emitter-node new-emitter-location)
   :ACTION-PART
   (PROGN ($SUPERSEDE new-emitter-node
              ((emitter-my-location
                (CU:SHARED-LIST new-emitter-location)))
              'new-emitter
```

```
            'process-observation-2-new-nodes)

        ($SUPERSEDE the.observation
            ((observation-associated-emitter
              (CU:SHARED-LIST new-emitter-node)))
            'emitter-matched)
            '[process-observations-2-new-nodes))

        ($MODIFY new-emitter-location
            ((emitter-location-control-information-site
            (LIST observation-site
                observation-time the.observation)))
            'new-emitter-location)
            'process-observations-2-new-nodes)))
  :RULE-CONTROL (:lhs t :rhs nil)
)
```

D.3. Control Information

```
(setq *Initializer* 'ELINT-INITIALIZATION)

(setq *Termination-condition* 'ELINT-QUIT)

(setq *Postprocessor* 'PRINT-RESULTS)

(setq *Event-extractor* (function $ALL-SORT))

(setq *Expectation-extractor* (function $FIFO))

(setq *Goal-extractor* (function $FIFO))

(setq *Cage-controls* '(:serial))

(defvar *agenda*
      '(clock-tick cleanup-cluster new-observation-read
                   corrected-emitter corrected-emitter-location
                   new-emitter-location
                   matched-emitter-location new-emitter
                   computed-fix
                   earlier-fix-computed associate-emitter-cluster
                   emitter-cluster-already-matched
                   assoc-cluster-emitter new-cluster
                   matched-cluster))

(DEFCONTROL-RULE  :CRULE-1
  :condition (not (equal (send EXPECT-INFO :STEPLIST) nil)
  :step :EXPECTATION)

(DEFCONTROL-RULE  :CRULE-2
  :condition (not (null (control-events)))
```

```
  :step :EVENT)

(setq *input-data-file* "s2:>aiello>basic.input")

(defvar  *output-trace-file* "local:>aiello>trace-elint-on-care.lisp")

(defvar *time-trace-file* "local:>aiello>trace-times.lisp")
```

D.4. User Functions

```
(DEFUN TIME-OF-INPUT-RECORD
  (RECORD)
  ""
  (DECLARE (UNSPECIAL RECORD))
  (if (equal record 'eof) nil (first RECORD))
)

(DEFUN INPUT-PROCEDURE
  (RECORD TIMESTAMP)
  ""
  (DECLARE (UNSPECIAL RECORD TIMESTAMP))
  (IGNORE TIMESTAMP)
  ($create   OBSERVATION  (make-observation
                          TIME   (cu:shared-list (FIRST RECORD))
                          SITE   (cu:shared-list (SECOND RECORD))
                          EMITTER-ID (cu:shared-list (THIRD
RECORD))
                          LOB  (cu:shared-list (FOURTH RECORD))
                          OBSERVATION-TYPE (cu:shared-list
                                                (FIFTH RECORD))
                          MODE (cu:shared-list(SIXTH RECORD))
                          SIGNAL-QUALITY (cu:shared-list
                                                (SEVENTH RECORD))
                          REDIRECT-FLAG  (cu:shared-list
                                                (EIGHTH RECORD)))
           nil  (quote NEW-OBSERVATION-READ)  (quote INPUT) )
)


(defun elint-initialization ()
  ($create report-data-level (make-report-data ) nil  'report-node
           'elint-initialization)
  ($create emitter-manager-level (make-emitter-manager ) nil
       'emitter-manager-node 'elint-initialization)
  (output-if *debugging-cage*  "~&Opening trace file ~A" *output-
           trace-file*)
  (when *output-trace-file*          ; [nps]
```

```
    (setq *output-trace-stream* (open *output-trace-file* :direction
                                      :output))
    (cu:without-clock (print (list ':data-rate bb:*domain-ms*)
                             *output-trace-stream*)
          (print (list ':circuit-name user:*circuit*)
            *output-trace-stream*))
    (output-if *debugging-cage*  "~&Opened trace file ~A"
            *output-trace-file*))
   (when cu:***count-locks***
    (setq cu:*count-locks-stream* (open *count-locks-file* :direction
                                       :output)))
   (when  ***time-trace***
    (setq *time-trace-stream* (open *time-trace-file* :direction :
                                    output))))

(defun elint-quit ()
  ;(print "testing termination-condition" cu:*output-stream*)
  (cu:without-clock (and (cu:shared-read *end-of-input*)
            (null (control-events))  ;former event list
            (or (cu:wait 100) t)
             (null (control-events)))))

(defun PRINT-RESULTS
       nil
       (cu:without-clock (when *output-trace-file*
                 (prin1 "elapsed time in mins. = "
                              *output-trace-stream*)
                 (print  (quotient (quotient (time-difference (time)
                                             b:*simulation-start-time*)
                                          60) 60)
                                *output-trace-stream*)
                 (close *output-trace-stream*)
                 ;Print results of Elint computation
                 (mapcar (function DESCRIBE-FLAVOR)
                              (send  (send(HYPOTHESIS-STRUCTURE)
                                          :cluster-level) :NODES))
                 )
                (fs:close-all-files)
                ))

(defun $ALL-sort (change-list-address &aux local)
  (unwind-protect
     (progn
      (cu:shared-lock (locf (control-lock)))
      (setf local (cu:shared-read change-list-address))
      (cu:shared-write change-list-address nil))
    (cu:shared-unlock (locf (control-lock))))
  (agenda-sort (cu:cache-shared-list local)))
```

```
(defun Agenda-compare (event1 event2)
  (declare (special *agenda*))
  (let ((type1 (change-type event1))
        (type2 (change-type event2)))
    (let ((pos1 (or (cl:position type1 *agenda*) 0))
          (pos2 (or (cl:position type2 *agenda*) 0)))
      (greaterp pos1 pos2))))

(defun agenda-sort (event-list)
  (sort event-list #'(lambda (ev1 ev2)  ;do we need sort-stable here?
                       (agenda-compare ev1 ev2))))
```

Knowledge Systems Laboratory
Report No. KSL-89-43

January 1991

An Application in Poligon

by
Jean-Christophe Bandini
and
James Rice

Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA 94304

The authors gratefully acknowledge the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

Abstract

This paper describes the design, implementation and performance of ParAble, a problem solving application built using the Poligon framework, a concurrent blackboard-based programming system. ParAble is a system for the diagnosis of faults in particle accelerator beamlines. The factors that motivate and constrain the design of Poligon applications are discussed. Experimental results and their interpretation provide an evaluation of the Poligon system in terms of the performance of this application.

1. Introduction

Concurrent problem-solving is a relatively recent field that has emerged as a result of falling hardware prices and the consequent burgeoning availability of multiprocessors. With the availability of concurrent hardware has come the problem of programming these machines. Multiprocessors have been used for quite a while for very regular, primarily numerical problems. The work described in this paper was performed within the Advanced Architecture Project (AAP) of Stanford University's Knowledge System Laboratory [Rice 88b]. The goal of the AAP is to investigate the use of multiprocessors for AI applications in the hope of achieving substantial speedup due to parallelism. Why this is not a trivial problem is described in [Rice 88a].

The goal of this sub-project was to study the design and performance of a new application mounted on Poligon [Rice 86] and [Rice 89], a concurrent problem-solving framework developed as part of the AAP. This application was required to be different from those which had already been studied (real-time radar signal interpretation [Nii 88]). The main issues of interest were:

- Performance measurement
- Measurement and analysis of speedup
- Study of resource allocation
- Study of the influence of granularity on system performance
- Design: the adequacy of Poligon for a different type of programming problem and the influence of the requirements of concurrent problem solving on the conceptualization of the problem and the knowledge engineering process. Previous work on the AAP had already determined that an appropriate decomposition of the problem domain into suitably parallel terms is a major part of the problem of concurrent problem solving.

Although we needed a "new" application, instead of starting from scratch, it seemed better to select an existing, serial, application and implement it using Poligon. This approach would let us draw comparisons between the serial and parallel implementations in term of design. The application we chose was called ABLE which had previously been developed by Scott Clearwater and Larry Selig [Selig 87] to help to align and debug particle accelerator beamlines.

The experiments described in this paper were carried out on the CARE simulator [Delagi 86a], which can be used to construct simulated multiprocessors of various shapes and sizes with a set of instruments to analyze the run-time behavior.

1.1. The Target Problem Solving Framework: Poligon

The Poligon framework is a skeletal blackboard-based concurrent problem solving system. Unlike conventional blackboard systems, to make efficient use of the multiprocessor hardware for which Poligon was designed, there is no centralized control.

The nodes of the blackboard can be viewed as active agents with attached daemon-like rules triggered by changes made to the blackboard nodes. When a node is created it is installed on a processor-memory pair (i.e. a processing element of the CARE machine) and rule invocation may use other processor-memory pairs (see Figure 1). The allocation of blackboard nodes to processors is usually handled by the Poligon framework, though it can be influenced by the application program.

Poligon is a high level approach to parallel programming and a lot of details are taken care of by the system, such as node object instantiation, rule invocation and the creation and defuturing of futures. By default, Poligon tries to parallelize as much as possible in the user's application, but the programmer can place restrictions on this parallelization when more control is required.

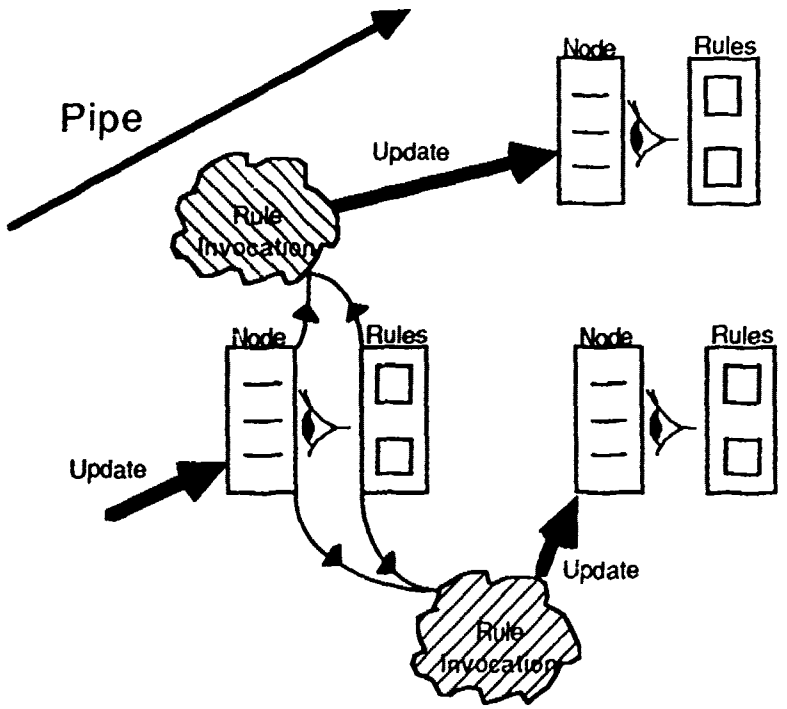


Figure 1. Poligon blackboard nodes and rule invocation. Updates to slots in Poligon nodes cause the concurrent activation of rules associated with those slots. These rules in turn go on to cause updates to slots in other nodes. This results in the implicit creation of pipelines.

2. The Application

The ABLE application was developed to solve the problem of misaligned or defective beamlines in particle accelerators. Beamlines are composed of a source of particles and a target connected by a pipe (see Figure 2). The particles are guided through the pipe with magnets which can focus, defocus and bend the beam very much like lenses and prisms affect rays of light. Monitors distributed along the beamline provide data to tune the mag-

nets. The tuning operation is done on-site with the beam on. This mode of operation entails high costs (often of the order of $0.25M per day) and requires considerable expertise and can often still take of the order of months to complete. The goal of the ABLE project was to automate the process of finding faulty magnets and monitors by using expertise, an analytical model and simulation.

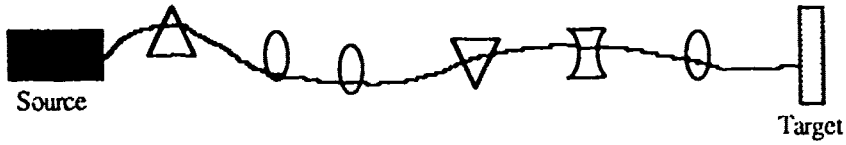


Figure 2. A Beamline. In particle accelerator beamlines, magnets are used to serve functions analogous to focussing, defocussing and prismatic components in an optical system. The source emits sub-atomic particles whose behavior is ideally to be measured by detectors sited at the target. The path of the beam, even in the absence of magnets is roughly sinusoidal because of quantum mechanical effects.

In order to control the beam as it travels along the beamline, monitors (detectors) are placed in the beam pip., which can detect the proximity of the beam. The output of these monitors can be used to control the strengths (and hence "refractive indices") of the magnets and thus control the beam. Unfortunately, physical limits prevent beamline designers from putting monitors in all the ideal places. This is often caused by the fact that magnets are physically so close together that there is no room for a monitor. Thus, the beamline debugging process must reason from incomplete information (see Figure 3).

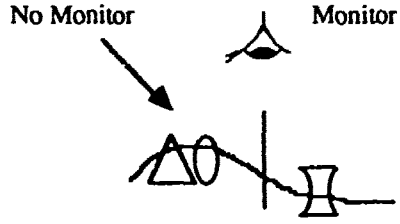


Figure 3. Monitors are placed along the beamline where feasible.

2.1. Serial Problem Solving

The method used to solve the beamline alignment problem at most existing particle accelerators can hardly be called anything other than mere knob-twiddling. Those trying to commission the beamline tweak the strengths of the magnets in the hope of achieving better alignment. This can often take many months because the effect of adjusting a particular magnet's strength can be strongly counter-intuitive. This counter-intuitivity can be due to many number of factors. For example, the effect of a change to a particular magnet's behavior is often not visible until the fractional change in beam trajectory it causes has been integrated down a substantial portion of the beamline. What is more, because sometimes the beamline is physically incorrect, either because someone bumps a truck into a magnet, deforming or moving it, or because magnets and monitors can easily be wired up with their intended polarity reversed, those commissioning the beam can exhaust vast resources in assuming that the problem can even be solved by parameter adjustment.

In response to these primitive debugging methods the ABLE system was designed. The serial ABLE problem solving method used numerical simulations and expert knowledge to

find the beamline error as quickly as possible. The numerical simulations provided two kind of results:

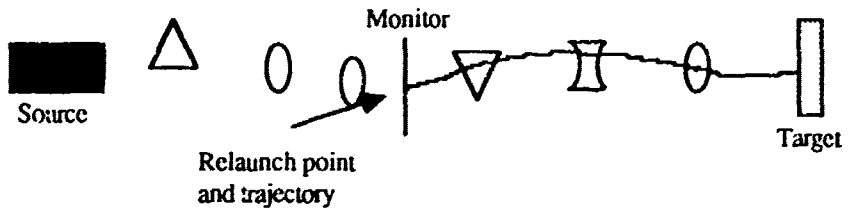


Figure 4. The simulated beam can be relaunched from any point in the beam. If it is relaunched with a trajectory equal to that measured from the real beam at the specified relaunch point then a faultless beamline would result in a perfect match between the trajectories of the simulated and real beams downstream of the relaunch point.

- *Relaunching*: the analytical model of the beam can be started at any point in the beamline with the beam being started with a known trajectory (see Figure 4). This allows the simulation of any segment of the beamline under controlled conditions. Because the real-world beam's trajectory can be determined by the monitors in the beamline, a simulated relaunch can be made from the position of a monitor with a trajectory equal to the measured trajectory. Comparison of the real beam with its simulated beam's path downstream of the relaunch point allows the system to deduce the approximate location of errors (see Figure 5). The exact location cannot easily be found because some errors do not show up until a long way "downstream" and also because the number of monitors is not as great as would be ideal because of physical limitation in the construction of the beamline.

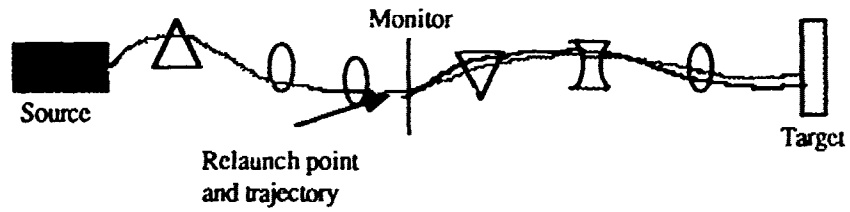


Figure 5. In this case the simulated beam rapidly diverges from the real beam's measured path (gray). This indicates that a fault is likely near the monitor from which the relaunch was made.

- Magnet Fitting: Once the approximate location of an error has been found, a linear optimization process can be performed, which modifies the parameters of the possibly erroneous magnets around the suspected error point until a good fit is observed between the simulated and the actual beam paths.

The serial problem solving method relaunches the simulated beam from the beginning of the beamline. A set of rules is used to analyze the differences between the simulated and real beams and also to find at which monitor the error is located. This monitor is called the "bad-monitor"[1]. A range of monitors downstream of the "bad monitor" constitutes a region of the beamline which is supposedly error-free and which is called a "good-region". The beam is then relaunched from the end of the good-region to find the next bad-monitor and the process proceeds to the end of the beamline.

[1]Note: This does not indicate that the monitor is broken, though it may be, it simply identifies a possible, approximate location for the error.

The magnets in the vicinity between good-regions (i.e. around a bad-monitor) are then checked to see if, by varying their parameters, they can account for the observed discrepancy between real beam data and the simulated beam path. In the best of all possible worlds the beam alignment problem could be solved by running an optimization process over all of the parameters of all of the magnets in the beamline simultaneously. Unfortunately, this is combinatorially explosive, so the goal of the ABLE system was to localize the area of the problem so that the computational task of performing these linear optimizations was tractable.

The serial implementation of ABLE runs using KEE™and a large body of FORTRAN code to perform the simulation of the beam and the above mentioned linear optimization.[2] ABLE was tested on real data from the Stanford Linear Accelerator (SLAC) and was shown to be able to find problems in only a few minutes that would have required days or possibly weeks to find by traditional means.

2.2. Parallel Problem Solving

ParAble is a parallel implementation of an ABLE-like system. In many senses ParAble is an artificial problem since the commissioning of beamlines is something that takes place over months, and it is really not necessary to gain any speedup over the existing ABLE implementation. However, recent developments in particle accelerators, particularly those spurred by the Strategic Defense Initiative, has resulted in designs for accelerators that are much more sophisticated and in need of much more automation so as to control and debug them. It is by no means ridiculous to think in terms of accelerators that would require continuous real-time monitoring and debugging so as to keep them running at peak performance.

Nevertheless, independent of the requirement for a real-world parallel implementation of ABLE, our own goals were to investigate the process of concurrent problem solving both from the human and the machine's point of view and in this respect the ParAble application was thoroughly instructive. The design and problem solving strategies in the serial and parallel systems are widely different. This section provides a high-level description of the design of the ParAble application.

2.2.1. Goals

The goal of the new design for a parallel ABLE was to find a reformulation of the problem solving method which would make efficient use of the underlying parallel architecture. To this end it was our goal that the expert, Scott Clearwater, an accelerator physicist, should try to reformulate the problem solving method so that:

- The problem was solved by solving independent subproblems, so that the computation could be split between several processors without requiring synchronization and communication.
- The problem solving method exploited parallelism whenever possible.
- The need for control in the problem solving method was reduced to a minimum.

The motivation for these goals was the efficient use of parallel hardware with minimal synchronization and communication.

[2]KEE is a trademark of IntelliCorp Inc.

2.2.2. Problem Solving

The problem solving method used in ParAble differs from that used in the serial ABLE implementation in a number of interesting ways. In the serial ABLE implementation, a single relaunch of the simulated beam is made from the beginning of the beamline to locate the monitor where the actual beam and the simulated, relaunched beam diverge, this monitor is termed a "bad monitor". When asked to think of how he would ideally solve the problem, our expert came to the conclusion that relaunching the beam only once to find the likely location of the error was not at all the best way to think about the problem. This was because using only one simulated relaunch of the beam made it very difficult to diagnose problems due to multiple errors in the beamline. In the case of ParAble, therefore, because the Poligon programming model assumes the availability of substantial computing resources, multiple, simultaneous simulated relaunches were possible, one from each monitor in the beamline, i.e. one relaunch for each available beam trajectory data point. Having multiple relaunches also improved the reliability of the system's conclusions and succeeded in getting correct diagnoses in some cases for which a single relaunch, such as that used by ABLE, would have resulted in a gross error concerning the location of the bad monitor.

In the serial implementation, even when relaunching is performed as part of the magnet parameter linear optimization, the simulated beam is analyzed only downstream of the relaunch point. This is because the expert was thinking originally of how he would address the problem serially. When debugging a beamline serially it makes sense to work downstream, since as you go you can be sure to what extent errors are influencing the propagation of the beam. This mind-set was also largely motivated by the fact that the real beam travels from one end of the beamline to the other, you cannot make the beam travel backwards in time. However, when the expert viewed the problem as one in which problem solving activity could occur concurrently he made the discovery that the real goal was to separate the beam into good and bad regions. It did not matter in which order this happened. What is more, because the beamline simulator simulates the magnets in terms of their transfer function by means of matrix operations, the beam is simulated as a mathematical abstraction, not as a discrete simulation of the propagation of individual particles. Thus, the simulation *can* be run "backwards", i.e. it proved to be entirely legitimate to correlate the relaunched beam with the actual beam both upstream and downstream of the relaunch point. This means that a relaunch from any given monitor position can suggest the presence or an error, or lack thereof, either upstream, or downsteam of the relaunch monitor, or both. This reconceptualization of the problem allowed a totally different problem solving method from the serial ABLE implementation. Rather than relying on only one relaunch to find an error, the system was able to relaunch the beam from every monitor and use the results of analyzing each relaunch in a voting scheme to pick the most likely cause(s) for the misalignment.

Thus, the parallel problem solving method is organized in two major steps. First, find the "bad region(s)" along the beam. As mentioned above, a bad region is a segment of the beamline between some sequence of monitors, which presumably has a faulty magnet. Since several relaunches are performed, their conclusions need to be integrated to find where the bad regions are. This is done by a voting scheme. The second phase in the problem solving activity required that for each bad region the system we should find the bad magnet (the magnet causing the actual beam misalignment). Once a bad region is known to have a faulty magnet, linear optimization runs can be performed for each of the magnets in the bad region simultaneously to find the one at fault.

2.2.3. Sources of Parallelism

From a problem solving point of view, there are only really three sources of parallelism. Poligon is designed to be able to exploit these on distributed memory multiprocessors.

- Pipeline parallelism. This is the form of parallelism seen on industrial assembly lines. The amount of speedup in a perfectly balanced pipe is proportional to the number of stages in the pipe. If the ParAble application were to have been used in a continuous, real-time manner, then we could have hoped for pipeline parallelism as a result of pumping new data into the system while it was still working on old data. We did not, however, investigate this area, since real-time systems was the primary research area of other parts of the Advanced Architectures Project.
- Replication. This is the parallelism due to having multiple processors all doing similar things to different data. This form of parallelism is more like that seen in a car repair shop, where there is likely to be one mechanic working on each car. More speedup for the business overall could be achieved by adding more mechanics and getting more cars to work on. This is the form of parallelism most appropriate to the ParAble application. Indeed, the presence of multiple magnets and monitors in the beamline, each of which could be considered in the problem solving process indicates that one might hope for speedup that was proportional to some function of the number of magnets and monitors.
- Decomposition into separate sub-problems. Some might view this aspect as being no different from the two previous ones, that is, a pipeline represents a decomposition of the problem too. We include this since there are often qualitatively different things that have to be done, which can nevertheless be done in parallel. This could be viewed as replication at some level of abstraction, but such a view does not help the cognitive process of problem decomposition. For example, when building a house, it is possible to install the plumbing at the same time that it is being wired and roofed. Clearly, each of these activities is being done by a similar "processor", but it is not useful to think of them as being simply replicated, since at the house building level of abstraction they are still qualitatively different operations. We can think of this form of parallelism as "Knowledge Parallelism"

The parallelism that results from replication is often referred to as "Data Parallelism", since it is the form of parallelism that is a function of the structure of the data in the problem, not the processing that has to be done on the data. Adding more data typically adds more potential for parallelism. The main source of parallelism in ParAble is data parallelism.

- During the finding of the bad region(s) multiple relaunches must be run. These relaunches can be run in parallel and they do not require any synchronization.
- Once the bad region(s) have been found, multiple magnet fit simulations (linear optimizations) must be run. These optimizations can also be run in parallel.
- Overlapping between the two phases is possible (pipelining). Once a bad region has been found the finding of the faulty magnet is an independent subproblem which can be solved concurrently with relaunches or other bad region finding subproblems.

Clearly, the numeric simulations and optimizations may well also offer considerable opportunities for parallelization. However, our project was more interested in the process of concurrent symbolic programming, than numeric programming. Thus, we chose to view these activities as monolithic (black boxes). We were, however, able to adjust the simulated time taken to execute the simulation. This was possible for our experiments by executing the simulation for every possible combination of parameters and measuring the runtime of each such execution. Then, when ParAble wanted to execute a simulation it had, in fact, only to look the result up in a table and charge the appropriate amount of time to the

CARE simulator. This strategy allowed us to investigate the impact of the speed of the beamline simulations on the overall performance of ParAble.

2.2.4. Design of ParAble for Poligon

Poligon's programming model gives the user a view of the world that is separated into objects that belong to classes. These classes represent the natural partitions in the solution space, often referred to by blackboard systems as "levels" because they are often used to represent distinct levels of abstraction in the solution space. Knowledge in the form of pattern/action rules is associated with these classes and hence with their instances. The design of ParAble uses Poligon nodes to represent and hold the state of the beamline objects and the state of the evolving solution. More specifically, the classes of Poligon nodes used in this application were as follows (see also Figure 6).

- Magnet, instances of which hold the state of the real magnets of the beamline.
- Monitor, instances of which hold the state of the real monitors.
- Segment, instances of which hold the state of the region of the beamline delimited by two consecutive monitors.
- Experiment, of which there is only one instance which retains the overall state of the solution and which is used for some global synchronization and initialization.
- Bad region, whose instances represents a sequence of monitors that contain a magnet error.

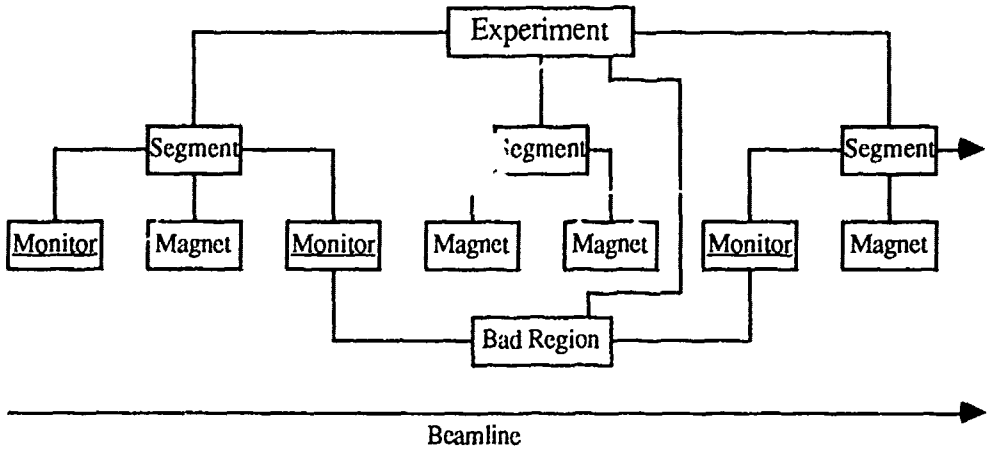


Figure 6. The configuration of ParAble's blackboard.

2.2.5. The Application in Operation

ParAble's design was strongly organized so as to exploit the sources of parallelism described above. Its behavior fell into two primary components, the finding of the bad region(s), and the finding of the bad magnet within each of these regions.

To find the bad regions, a simulation was relaunched in parallel for each monitor. The integration of the simulation results used a voting scheme, whose goal was to reduce any synchronization overhead to a minimum. The result of each relaunch was used in a distributed fashion: a vote was sent, by the monitor that performed the simulation, to each of the monitors which were suspected as being bad as the result of the simulation. Each monitor collected the votes cast for it and was empowered to make a decision on whether to create a bad region or not on the basis of the votes cast. A simple comparison of the sum of the votes with a threshold proved to be enough, but further refinements may be possible. This scheme avoided any bottleneck that might have been caused if a central object had

been used to collect all the results of the simulations and to make a final decision. Furthermore, this design allowed the finding of bad regions before the completion of all the relaunches by all of the monitors, i.e. there was no synchronization necessary in order to continue with the problem solving. Nevertheless, some control was necessary after the decision to create a bad region was made to avoid creating overlapping bad regions. This would have entailed redundant computation.

To find the bad magnet within a bad region, ParAble started by creating a Bad Region object. This bad region was then responsible for solving the subproblem of finding the bad magnet inside the region of the beamline that it denoted. Linear optimizations were run in parallel for each magnet in the bad region so as to try to find a set of magnet parameters that would best fit the behavior of the real beam. All the results of these optimization operations had to be collected before making a choice concerning the bad magnet.

3. Experiments on ParAble

Numerous experiments were performed on Poligon and ParAble, some of which we will describe here. As was typically the case in experiments on the AAP, our primary concern was for speedup. We were, however, also interested in using the experiments to deliver some insights concerning the generality of Poligon and the probable limits of its performance. Thus, the main goals of these experiments were:

- Measurement of speedup.
- Study of any resource allocation problems.
- Study of the performance of ParAble when encountering multiple errors as opposed to only one error in the beamline. Note, the serial ABLE implementation was not able to handle multiple errors satisfactorally at all.
- Study of the granularity of Poligon and the application.
- Validation of previous experimental results.

3.1. Experimental Parameters

The design of the program was kept constant for all experiments. The parameters that were changed for the experiments were:

- The data set: a variety of data sets were available. These fell into two broad categories: those with single errors in the beamline and those with double errors.
- The numerical simulation timing scale factor: Simulations were involved whenever a relaunch was made or whenever an optimization run was made. A scale factor was applied to the true, wall-clock time of these simulations in order to study the behavior of the system with respect to computational grain sizes.
- The number of processors: These ranged between one and 128 in powers of two.

3.2. Experiment Measurements

The experiments measured the execution time of the application. More precisely we measured:

- Initialization time: the time to set up the Poligon objects.
- Problem solving: the time to solve the problem, (i.e. total time minus the initialization time). This time was broken up into two main quantities:
 - the time to find the bad-region
 - the time to find the bad-magnet inside the bad-region.

For a finer grained analysis we also use a time stamped trace of the executions. In the following experiments we did not pay much attention to the initialization time. This was be-

cause we anticipate that if a system such as ParAble were to be used in the real world it would be used in a real-time manner. In this case, initialization is payed for only once at load time and is therefore not relevant to normal system operation. In these experiments, the initialization time was not trivial because of the time taken to create the objects on the blackboard and to connect them up in a manner suitable to the application. This involved a certain amount of synchronization.

3.3. Theoretical Analysis

The first question to answer is: what is the available parallelism of the application and what is the maximum speedup we can expect? As mentioned in the description of the program design, the two main sources of parallelism are:

- Multiple relaunch simulations can be carried out in parallel.
- Once a bad region has been found, the computation of the linear optimization to find the bad magnet can be performed in parallel.

Let us assume we have infinite resources, infinite Poligon system speed and instantaneous communications. The execution time of the relaunch simulations to find the bad magnets all have about the same duration. The relaunch time $T_{relaunch} \approx 2$ seconds using a simulation scale factor SF=1, except for one which takes about 4 seconds. Since 17 relaunches are typically performed, the theoretical speedup for the relaunches is:

$$Speedup_{relaunch} = \frac{(16T_{relaunch} + 2T_{relaunch})}{2T_{relaunch}} = 9$$

But the speedup for finding the bad region is different because bad region finding does not require waiting for the results of all the relaunch simulations, since a subset may be enough to go above the vote threshold. In particular, the long simulation ($2T_{relaunch}$) may not be necessary, thus, in general:

$$Speedup_{bad\text{-}region\text{-}finding} = \frac{(16T_{relaunch} + 2T_{relaunch})}{T_{relaunch}} = 18$$

For the magnet optimization simulations, the bad region has 8 magnets and the average simulation time is $T_{opt} \approx 2.5$ seconds but one of the simulation times is 4.1 seconds (= $1.65T_{opt}$). Thus, the maximum speedup for the magnet optimization simulation is:

$$Speedup_{magnet\text{-}opt} = \frac{(7T_{opt} + 1.65T_{opt})}{1.65T_{opt}} = 5.25$$

Thus for the overall problem-solving speedup we have:

$$Speedup_{total} = \frac{(16T_{relaunch} + 2T_{relaunch} + 7T_{opt} + 1.65T_{opt})}{(T_{relaunch} + 1.65T_{opt})} = 9.4$$

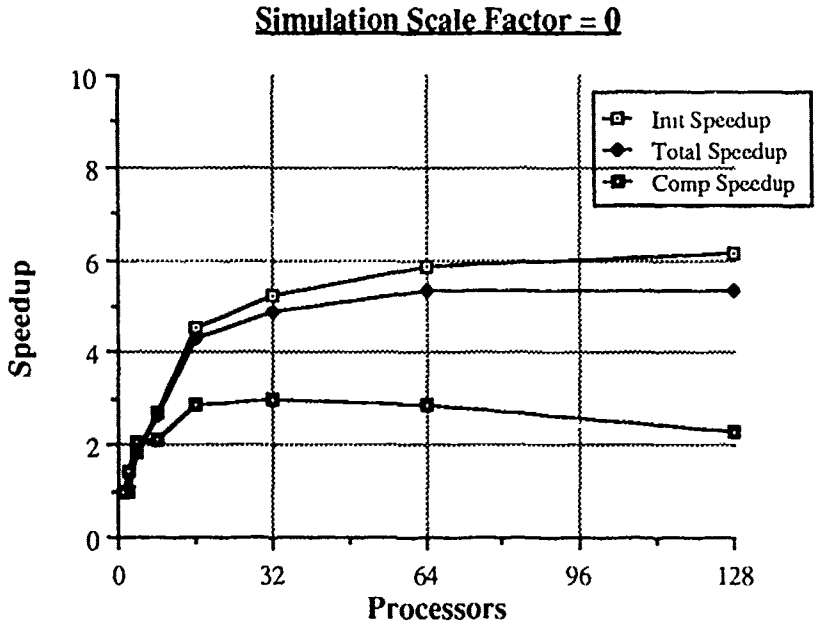
In summary:

Speedup source	Maximum Theoretical Speedup
Bad region finding	18.0
Bad magnet finding	5.25
Total	9.4

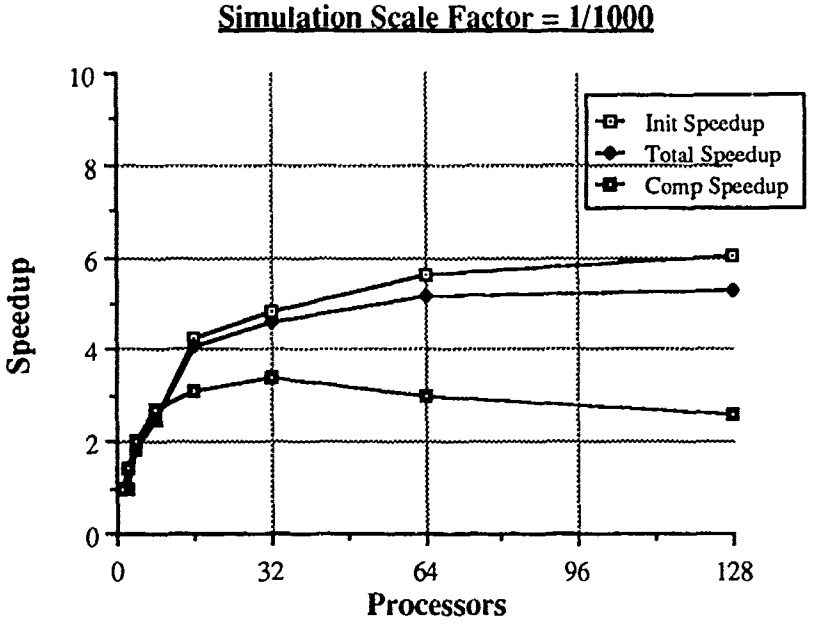
3.4. Measurement of Speedup

The purpose of this experiment was to have a coarse approximation of the speedup and to have a basis to analyze the performance. This experiment was run with a single error data set. The scaling of the times for the simulations used the scale factors; 0, 1/1000, 1/100, 1/10 and 1, relative to the actual run time of the beamline simulation when executed in FORTRAN on a Lisp Machine. Thus, one data point represents what would happen if the simulations ran infinitely fast (0), and another data point refers to the simulation running in one tenth (1/10) of the actual measured time. The reason why all of our measurements used scale factor less than or equal to one was that we knew that the simulations ran at at least this speed on a real machine. The simulator could probably have been made faster by better programming, faster hardware or by the use of parallelism. There was, therefore, no reason ever to suspect that this code would run slower than its measured performance. This experiment was performed on eight different sized processor networks comprising respectively, 1, 2, 4, 8, 16, 32, 64 and 128 processors.

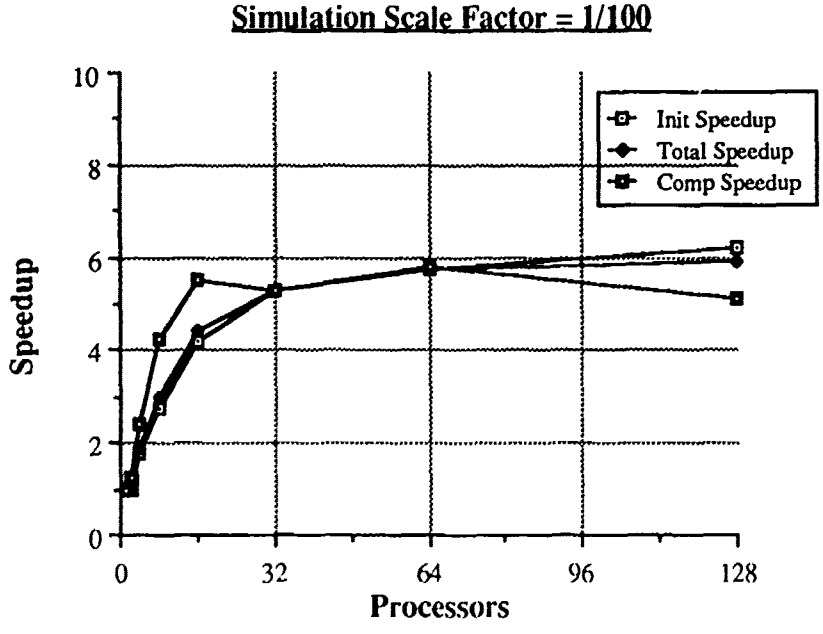
The results from these experiments are shown in Graphs 1–5.



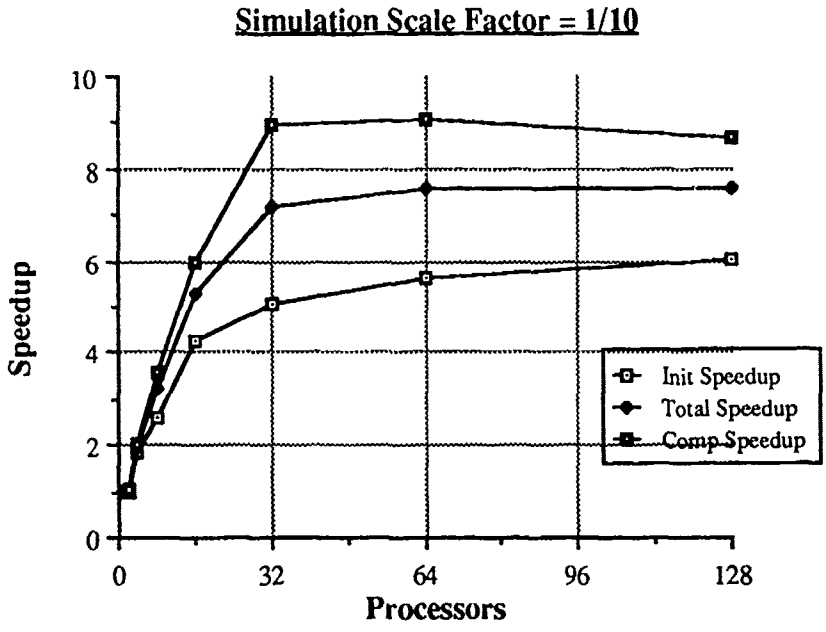
Graph 1. *Speedup of the ParAble application measured with no time at all spent in the beamline simulator.*



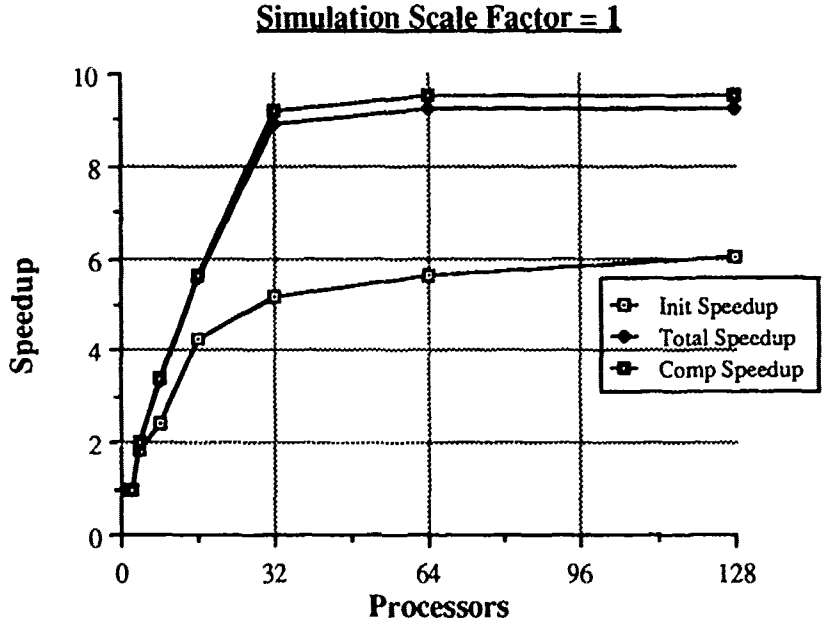
Graph 2. Speedup of the ParAble application measured with the time spent in the beamline simulator being only 1/1000 of the real-world time needed for the simulations.



Graph 3. Speedup of the ParAble application measured with the time spent in the beamline simulator being only 1/100 of the real-world time needed for the simulations.



Graph 4. Speedup of the ParAble application measured with the time spent in the beamline simulator being only 1/10 of the real-world time needed for the simulations.



Graph 5. Speedup of the ParAble application measured with the time spent in the beamline simulator being equal to the real-world time needed for the simulations.

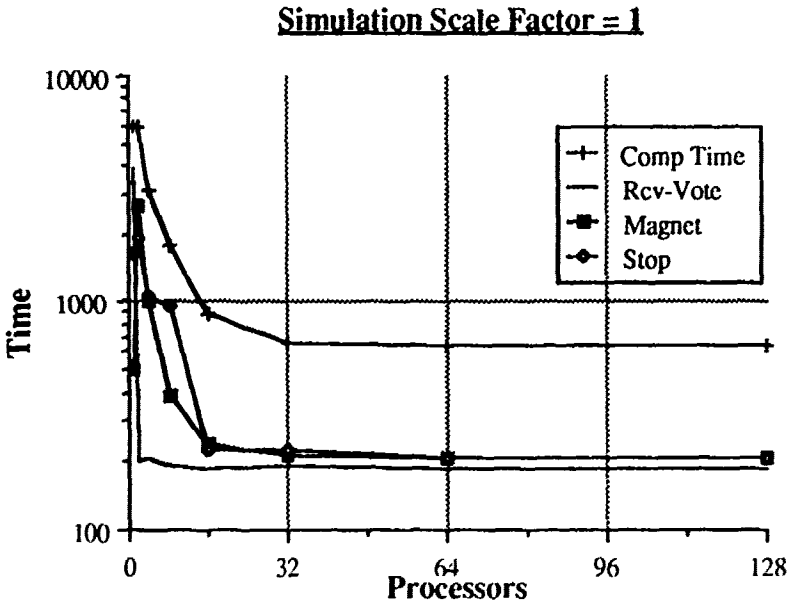
3.4.1. Interpretation

Each of the Graphs 1–5 have three curves. The curve marked "Init Speedup" is the speedup curve resulting only from the timing of the system's initialization. As can be eas-

ily seen from all of the graphs, the initialization of the application had a certain amount of concurrency, which was independent of the beamline simulation scale factor because, of course, there was no need to run the beamline simulator during the application's initialization process. The initialization procedure consistently delivered a speedup of about six. This is consistent with other results derived on the AAP, which have shown that speedup of the order of ten is relatively easy to achieve. Not much effort was spent in making the initialization more effectively concurrent, indeed, it was only made parallel at all because this is the natural way to program in Poligon.

The second curve is marked "Total Speedup" This indicates the speedup resulting from the whole of the application's execution, including initialization. We we mentioned above, the initialization time was not deemed to be as interesting as the problem solving aspects of the application, so this curve is shown mostly to give a feel of the effect of composing the two different components of the application.

The second, and most significant, curve on these graphs is labeled "Comp Speedup". This denotes the speedup delivered during the actual computation of the application. The speedup varied from a peak of about three for the scale factor (SF)=0 case to about ten in the SF=1 case. The speedup in the latter case was almost entirely due to the data parallelism inherent in the application. This can be seen most readily by examining Graph 6.

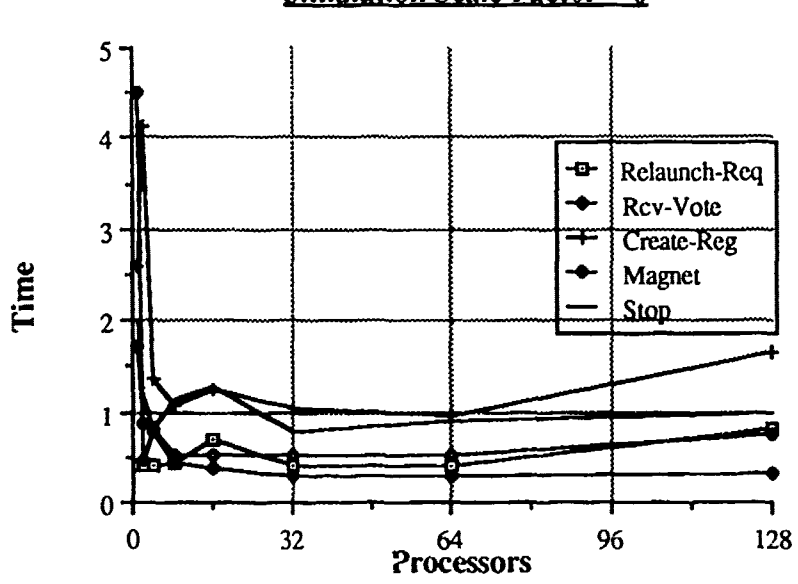


Graph 6. Execution times measured for different aspects of the ParAble application with simulation scale factor = 1.

In Graph 6, we see the execution times of different components of the ParAble application plotted against the size of the processor network used. The Y axis has a logarithmic scale to enhance the detail. One unit on the Y axis is equivalent to 100 CARE machine simulated microseconds. On this graph, we see the computation time, which is the same time used to compute the computation component speedup in Graphs 1–5, and the three components of the simulation that contribute most to it (they account for 99.5% of the time, at SF=1). These components are, respectively, the time taken to receive enough votes to be able to identify a bad region, the time taken to run the linear optimization process on the magnets in the bad region and the time taken to wait for the conclusion of the optimizations and to finish up. It turns out that the speedup in the vote receiving phase was about 18. This was

simply because of the number of monitors in the whole beamline (see theoretical discussion). The speedups derived in the other components were similarly a function of physical limits imposed by the structure of the beamline, not by the problem. The aggregate maximum speedup of about ten is simply a function of the fact that part of the time is spent in a highly parallel component, the bad region finding, and part is spent in the bad magnet finding component, which is less able to exploit parallelism because there are typically not many magnets in a bad region.

Thus, the speedup of the system, when not limited by the granularity of Poligon itself is determined entirely by the physical characteristics of the beamline. If we were to run on a larger, more complex beamline then we might reasonably expect to achieve more speedup.



Graph 7. The time taken to execute certain portions of the ParAble application for beamline simulation SF=0, plotted against the number of processors in the network.

If we now consider the diametrically opposite case, that of SF=0, we can see also how speedup is limited in the fine-grained case by the granularity of Poligon's rule execution mechanism. In this case, we see plots for the times taken by the vote receiving part, the magnet optimization and the stopping point as before, but in this case we also show two other typical times, one for the time taken to make the request to do a relaunch and one for the time taken to create a bad region. These times are entirely typical of those of other timed components in the ParAble system. From this we can conclude that, even if we pay no price for the beamline simulation, we still pay a price of about one millisecond for Poligon operations, such as rule invocation. This is consistent with the predictions and measurements made in [Nii 88] regarding the granularity of Poligon's rule invocation mechanism. We can therefore conclude from this that it is likely to be fruitless to break up a Poligon application into grains of less than one millisecond because the framework's overhead will result in an overall decrease in performance. The Amdahl limit for this application is therefore met early when the scale factor is set to zero, because there is a certain amount of processing that must be done serially, whatever. A discussion of the implementation of Poligon and of means by which this one millisecond overhead could be substantially reduced is given in [Rice 89]. Note that although we present here the experimental

results from only one data set, we in fact ran ParAble on a number of data sets and consistently achieved similar results.

3.4.2. Resource allocation

When we first ran the above experiment, we did not achieve the speedup reported. Furthermore the speedup plots that we received showed significant irregularities and we found the results not to be repeatable. We ran a number of additional experiments to determine why we were getting such irregular behavior from the system. Analysis of these experimental data revealed that the cause was Poligon's default random site allocation for processes. The Poligon model assumes that, by default, the computations being executed by the application are likely all to be approximately of the same duration and so, in the absence of a user specified resource allocation strategy, instantiates new blackboard nodes and executes rules concurrently on randomly selected processors. The rationale for this is that the "law of large numbers" will smooth things out in a large application. Unfortunately, the ParAble application is not like this, especially when the simulation scale factor approaches 1. In this case, the large computation grains are spawned and with Poligon's default allocation strategy, an analysis of the probability of a "collision" between large computational grains showed that in almost every run we would expect to have two long operations assigned to the same processor. The result of this was to double the apparent length of each of these components, thus increasing the serialization in the system and reducing parallelism. What we needed was a different resource allocation method.

What we chose to do was to divide the sites up in two equal sized groups, one group was dedicated to run Poligon rules that caused beamline simulations and the other group was used for everything else (blackboard nodes and other rule activation contexts). The allocation scheme we chose was round robin for the sites dedicated to the beamline simulations and random for the other sites.

For large beamline simulation granularity (SF=1), the following table shows that the speedups are very close to the theoretical ones we computed.

Speedup source	Theoretical Speedup	Measured Speedup (128 processors)
Bad region finding	18.0	17.5
Bad magnet finding	5.25	5.2
Total	9.4	9.5

Another important point to note is that, as can be seen in Graph 5, measured speedup leveled off above 32 processors. In other words, 32 processors is enough to reach the theoretical maximum speedup, when a careful resource allocation method is used.

3.5. Multiple Errors

The previous experiments were carried out with single error data set. In this experiment, we used a double-error data set. The experiment was in all other respects, basically the same as the previous experiment.

With multiple errors, a slight improvement in speedup could be expected for the magnet finding sub-task because with two regions, a larger number of magnet optimizations need

to be computed and thus the speedup is potentially larger. On the other hand, having two bad-regions means that each of the bad-monitors receives a smaller number of votes. Thus, the bad-region finding may take longer (although the total amount of relaunching is exactly the same as with a single error problem).

For the large granularity case (SF=1), the following table gives the results and a comparison with the results from the single error experiments.

Measured Speedup (128 processors)	Single Error	Double Error
Bad region finding	17.5	15.85
Bad magnet finding	5.2	5.75
Total	9.5	10.27

As can be seen easily from the above table, no significant change in speedup was observed. However, this belies the fact that ParAble was solving a problem that was twice as hard (and which the serial ABLE couldn't have solved at all). Thus, in some senses an extra speedup of a factor of two was delivered.

4. Discussion

We learned a great deal about using Poligon during the implementation of ParAble.

4.1. What is missing

Although the following features are available in Poligon by using programming tricks, it may be useful to integrate them in Poligon, or Poligon like architectures, for ease of use and eventually better performance:

- Rule locking, mutual exclusion between a set of rules. Other applications work in Poligon not described here has revealed that rule locking is sometimes necessary to enforce consistency between fields of an object.
- Rule on/off. In some cases it appears that it would be useful to have the ability to switch on/off a set of rules attached to a node. For instance, if some rules should become inapplicable after some event happens, this feature could be used. Poligon's expectation mechanism partially allows this sort of behavior, but not as first class behavior.
- Resource allocation. It would be useful if Poligon gave more support for controlling the mapping of objects and rule invocation contexts onto processors.

4.2. Programming Hints

Numerous lessons were learned about the programming process in Poligon itself. Here, we enumerate some of the programming tips we learned while implementing the ParAble application.

- The high level design should be done with two important ideas in mind. First, control should be reduced as much as possible because control entails synchronization, atomicity and communication overhead. Second, the blackboard, when viewed globally, has many transient inconsistencies, even in a well written application. The design of applications should take this fact into account.

- We found it useful to think of the nodes as objects and of the slots updates as asynchronous messages. This view of a Poligon program actually corresponds to the underlying implementation. It also avoids being misled by the usual assumptions we make when we deal with slot updates in a uniprocessor implementation of a frame system.
- The programmer should keep in mind the non-deterministic nature of the system.
- The Poligon system tries to parallelize as much as it possibly can by default. This characteristic implies that a lot of care should be taken to ensure data consistency when it is necessary, though Poligon's "smart slot" mechanism is helpful with this problem in general.

5. Conclusion

In this paper we described ParAble, an application program written to run on the Poligon concurrent blackboard architecture. ParAble is a concurrent version of ABLE, an expert system for the diagnosis of particle accelerator beamlines.

This project has shown that the Poligon framework can be effectively used for implementing problem solving systems other than real-time signal interpretation systems, such as Elint [Nii 88]. Speedup of the order of 10 could be achieved with careful resource allocation, further speedup being likely with a larger problem domain.

A number of experiments that were performed on ParAble were described and their results enumerated. These experiments highlight the significance of rule granularity and identify resource allocation as a crucial aspect of application design, particularly when computation granularity is heterogeneous.

6. Bibliography

[Aiello 86] Nelleke Aiello. User-Directed Control of Parallelism: The Cage System. Technical Report KSL-86-31, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986.

[Delagi 86a] Bruce Delagi. CARE Users Manual. Technical Report KSL-86-36, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.

[Delagi 86b] Bruce A Delagi, Nakul P. Saraiya, Gregory T. Byrd. LAMINA: CARE Applications Interface. Technical Report KSL-86-76, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.

[Delagi 88] Bruce A. Delagi and Nakul P. Saraiya. ELINT in LAMINA: Application of a Concurrent Object Language. Technical Report KSL-88-33, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.

[Engelmore 88] Robert Engelmore and Tony Morgan (eds.) Blackboard Systems. Addison-Wesley Publishing Company Inc., Menlo Park 1988.

[Ensor 85] J. Robert Ensor and John D. Gabbe. Transactional Blackboards. Proceedings of the 9th International Joint Conference on Artificial Intelligence: 340-344, 1985.

[Gabriel 84] Richard P. Gabriel, and John McCarthy. Queue-based Multi-processing Lisp. Proceedings of the ACM Symposium on Lisp and Functional Programming: 25-44, August, 1984

[Lesser 83] Victor R. Lesser and Daniel D. Corkill. The Distributed Vehicle Monitoring Testbed: A Tools for the Investigation of Distributed Problem Solving Networks. The AI Magazine, Fall:15-33, 1983.

[Nii 79] H. Penny Nii and Nelleke Aiello. AGE: A Knowledge-based Program for Building Knowledge-based Programs. Proceedings of the 6th International Joint Conference on Artificial Intelligence: 645-655, 1979.

[Nii 86] H. Penny Nii. Blackboard Systems. Technical Report KSL-86-18, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986. Also in AI Magazine, vol. 7-2 and vol. 7-3, 1986.

[Nii 88] H. Penny Nii, Nelleke Aiello and James Rice. Experiments on Cage and Poligon: Measuring the Performance of Parallel Blackboard Systems. Technical Report KSL-88-66, Knowledge Systems Laboratory, Computer Science Department, Stanford University, October 1988.

[Rice 86] James Rice. The Poligon User's Manual. Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.

[Rice 88a] James Rice. Problems with Problem-Solving in Parallel: The Poligon System. Technical Report KSL-88-04, Knowledge Systems Laboratory, Computer Science Department, Stanford University, January 1988. Also in Proceedings of Third International Conference on Supercomputing, May 1988.

[Rice 88b] James Rice. The Advanced Architectures Project. Technical Report KSL-88-71, Knowledge Systems Laboratory, Computer Science Department, Stanford University, January 1988.

[Rice 89] James Rice. *The Design of a High Performance, Concurrent Problem Solving System...and many Lessons Learned on the Way.* Technical Report STAN-CS-89-1294 (KSL–89–37), Heuristic Programming Project, Computer Science Department, Stanford University, November 1989.

[Selig 87] Lawrence J. Selig. An Expert System using Numerical Simulation and Optimization to find Particle Beamline Errors. Technical Report KSL-87-36, Heuristic Programming Project, Computer Science Department, Stanford University, 1987.

Knowledge Systems Laboratory
Report No. KSL 86-69

October 1986

An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures

by

Harold D. Brown, Eric Schoen, and Bruce A. Delagi

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

This research was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Eric Schoen was supported by a fellowship from NL Industries. Bruce Delagi is currently a visiting research scientist at Stanford from Digital Equipment Corporation.

Abstract

This report documents an experiment investigating the potential of a parallel computing architecture to enhance the performance of a knowledge-based signal understanding system. The experiment consisted of implementing and evaluating an application encoded in a parallel programming extension of Lisp and executing on a simulated multiprocessor system.

The choosen application for the experiment was a knowledge-based system for interpreting pre-processed, passively acquired radar emissions from aircraft. The application was implemented in an experimental concurrent, asynchronous object-oriented framework. This framework, in turn, relied on the services provided by the underlying hardware system. The hardware system for the experiment was a simulation of various sized grids of processors with inter-processor communication via message-passing.

The experiment investigated the effects of various high-level control strategies on the quality of the problem solution, the speedup of the overall system performance as a function of the number of processors in the grid, and some of the issues in implementing and debugging a knowledge-based system on a message-passing multiprocessor system.

In this report we describe the software and (simulated) hardware components of the experiment and present the qualitative and quantitative experimental results.

1. Introduction

This report documents an experiment investigating the potential of a parallel computing architecture to enhance the performance of a knowledge-based signal understanding system. This experiment was done within the Expert Systems on Multiprocessor Architectures Project of Stanford University's Knowledge Systems Laboratory.

The computational characteristics of complex knowledge-based systems are poorly understood, especially in parallel computational environments. Our Architectures Project is performing a number of experiments to try to gain some understanding of these characteristics and, in particular, of the potential for concurrent execution of such systems. A primary goal of the project is to develop software and hardware system architectures which exploit this concurrency to increase the performance of knowledge-based signal understanding and information fusion systems.

The Architectures Project is organized according to a hierarchy of computational abstraction levels as shown in Table 1-1. Each experiment represents a narrow, vertical slice through these levels and consists of a specific system choice for each level.

For the reported experiment, the choosen application is a knowledge-based ELINT (ELectronics INTelligence) system for interpreting processed, passively acquired radar emissions from aircraft. The ELINT application is implemented in CAOS, an experimental concurrent, asynchronous object-oriented framework built on Zetalisp [1]. The CAOS framework, in turn, relies on the services provided by the underlying hardware system environment. For this experiment, the hardware system environment is a simulation of a parallel architecture, called CARE [2]. CARE simulates a communications grid of processing sites where each site contains a Lisp evaluator, private memory, and a communications and process scheduling subsystem. Message-passing is the only means of inter-site communication. CARE is simulated using a general, event-based simulator, SIMPLE [3]. SIMPLE is written in Zetalisp and executes on a Symbolics 3600 or a Texas Instruments Explorer Lisp machine.[1] Figure 1-1 illustrates the relationship between the various software components of the experiment.

The ELINT-CAOS-CARE experiment investigated both qualitative and quantitative aspects of the performance of the overall system. The CARE architecture uses dynamic, cut-through (as

[1] A version of the SIMPLE simulator which runs on a local area network of multiple Lisp machines has also been implemented [4].

Table 1-1: Computational levels.

Level	Research questions
Application	Where is the potential concurrency in knowledge-based signal understanding tasks?
	How does the problem solver recognize and express application dependent concurrency?
Problem-solving framework	What are suitable framework constructs for organizing and encoding concurrent signal understanding tasks?
	What are appropriate granularities for knowledge, knowledge application and data to maximize concurrency?
	What types of strategies for control of knowledge application are needed to assure acceptable solution quality without introducing excessive execution serialization?
Knowledge representation and management	What kinds of knowledge representation mechanisms are suitable for exploiting concurrency in inference and search?
System programming language	How can general-purpose symbolic programming languages be extended to support concurrency and help manage the resource allocation and reclamation tasks on a distributed memory multiprocessor?
Hardware system architecture	What multiprocessor architectures best support the organization and concurrency in knowledge-based signal understanding applications?

opposed to store and forward) routing through the communication grid for interprocessor message transmission. Message transmission time is indeterminate. As a consequence, without the imposition of significant message sequencing protocols (and the corresponding serialization of execution), operations are intrinsically non-deterministic in the sense that two executions of the same program on the same input data can result in different problem solutions depending on different message arrival orders. For many knowledge-based systems, in particular, the ELINT system, there is no such thing as *the* correct problem solution but only *satisficing* (i.e., acceptable) problem solutions. One primary objective of the experiment was to investigate the trade-offs between the imposition of various synchronizations (and the resulting loss of concurrency) and the quality of the problem solution. A second primary objective was the more usual investigation of the speedup of the overall system performance as a function of the number of processing sites in the CARE grid. A third objective was to gain some understanding of the difficulties in implementing and debugging a reasonably complex knowledge-based system on a multiple address space, message-passing multiprocessor system such as that represented by CARE.

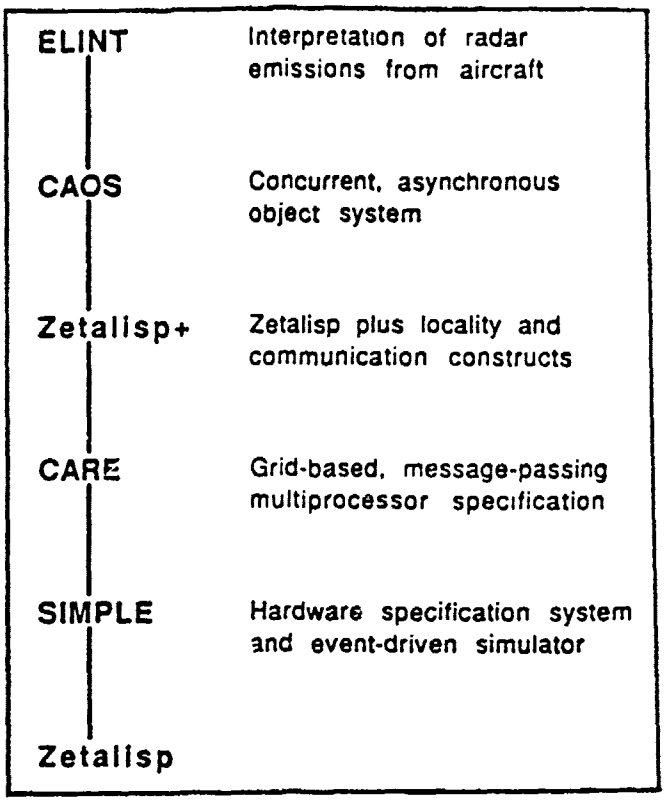


Figure 1-1: The software component hierarchy of the experiment.

In the following sections we describe, in decreasing hierarchical order, each component of the experiment. Section 2 describes the ELINT application. Section 3 gives an overview the CAOS programming framework and its approach to concurrency. ELINT's implementation in CAOS is described in Section 4, and Section 5 describes the salient features of the CARE architecture and its simulation environment. In Section 6 we present the results of the ELINT-CAOS-CARE experiment.

2. The ELINT Application

The driving application for our vertical slice experiment is a prototype, knowledge-based ELINT system for interpreting processed, passively acquired, real-time radar emissions from aircraft. This ELINT system is one component of a multi-sensor information fusion system, TRICERO [5] developed several years ago. ELINT was originally implemented in AGE [6], an expert system development tool based on the blackboard paradigm [7, 8]. ELINT is a relatively simple, but non-trivial, knowledge-based system. Much of its knowledge is implemented procedurally. However, if ELINT had been implemented as a production rule

system, we estimate that its knowledge base would consist of about one thousand rules.[2]

ELINT's basic analysis technique is to correlate a large number of passively observed radar emissions into the smaller number of individual radar emitters producing those emissions. It then correlates the emitters into the yet smaller number of clusters of co-located emitters. ELINT maintains the track and activity histories of the clusters

2.1. ELINT's Inputs

The inputs to the ELINT system are multiple, time-ordered streams of processed observations from multiple collection sites. Each observation is presented in a record format. The fields of an input observation record are shown in Table 2-1.

Table 2-1: Elint observation record.

Field	Contents
Observation-Time	An integer time-tag indicating when the radar emission was sampled
Observation-Site	The symbolic name of the collection site acquiring the observation
Site-Location	The positional coordinates of the collection site at the time of observation
Emitter-Identifier	An integer identifing the radar emitter producing the emission
Line-of-Bearing	The line of bearing from the collection site to the observed emitter
Emitter-Type	A symbolic radar emitter type designator
Emitter-Mode	The operational mode of the emitter at the time of observation
Signal-Quality	A symbolic indicator of the signal quality of the observed emission

The Site-Location field is necessary since the collection sites can be mobile. The Emitter-Identifier is a unique integer identifier assigned by the collection sites to each distinct observed emitter. This identifier is used by the collection sites to indicate multiple observations of the same emitter both over time and from different collection sites. In particular, two concurrent observations of the same emitter from different collection sites

[2] In general, there are currently no adequate metrics for measuring the complexity of knowledge-based systems. One crude measure used for rule-based systems is the number of rules. Although the number of rules does somewhat indicate the amount of knowledge, it does not give much indication of the complexity of the reasoning.

should have the same identifier. Both the intra-site and inter-site determination of whether two observed emissions are from the same emitter are based on the electronic characteristics of the emissions and on signature analysis. This determination may be in error, and the ELINT system must cope with such identifier errors. The **Emitter-Type** of a radar emitter indicates the functional class of the emitter, for example, Air-Intercept (AI), Navigation (NAV) or Identification-Friend-Or-Foe (IFF), and, if known, the equipment type class of the emitter. Certain classes of emitter types can have multiple operational modes. The **Emitter-Mode**, if applicable, is emitter-type specific. For example, an AI radar can be either in Search Mode or Lock-on Mode depending on whether it is scanning for a target or whether it is automatically tracking a specific target. The **Signal-Quality** of an observation is a subjective, qualitative measure of the strength of the observed emission, for example, *strong*, *normal*, or *fading*.

All of the input information required for the ELINT system is obtainable from the raw radar signal data using current, passive radar signal collection and processing techniques. These techniques are largely automated and employ special-purpose hardware.

2.2. ELINT's Outputs

The primary outputs of the ELINT system are periodic status reports about the tracks and activities of clusters of emitters in the area under surveillance. A cluster is defined as a collection of emitters which are co-located over time. That is, two emitters are in the same cluster if for some given minimum number of consecutive time units (three in the current ELINT system) their corresponding time-tagged locational fixes are within a distance determined by the line-of-bearing resolution of the observation site equipment (one degree resolution in the current ELINT system). Conceptually, two emitters are in the same cluster if if they are on the same aircraft or are on two tactically associated and co-located (over time) aircraft, for example, a lead aircraft and his wingman.[3]

The periodic output reports contain, for each cluster, information about the cluster's current

[3]An aircraft can be operating with some (or all) of its radars off. In general, it is impossible to distinguish between, for example, two co-located aircraft, one with an AI radar on and one with a NAV radar on, and one aircraft with both its AI and NAV radars on. Hence, our ELINT system does its assessments based on emitter clusters rather than aircraft.

heading, position and track; an estimate of the number and types of aircraft in the cluster;[4] an indication of the cluster's current activity; and an indication if the cluster represents an immediate threat, for example, if it is within a certain proximity of a friendly aircraft, if its AI radar is in Lock-on Mode, or if its missile guidance radar is on.

2.3. ELINT's Processing Flow

The basic reasoning strategy used by the ELINT application is data-driven accumulation of evidence for the existence, the tracks, and the activities of emitters and clusters based on input observations and infered information. The primary processing flow is a kind of pipeline where the pipeline stages are observations, emitters and clusters.

Upon receipt of a new observation, the system first determines if the observed emission *matches* (i.e., has as a source) a known emitter (i.e., an emitter on ELINT's "situation board"). This match is based on the **Emitter-Identifier** assigner by the collection site to the observation, and it is verified using the emitter's characteristics and its track and heading histories. Depending on the outcome of the match, one of the following actions is taken:

1. If the observation does not match a known emitter, then a new emitter which is the source of the observed emission is hypothesized on the situation board and initialized from the information contained in the observation.

2. If the observation does match an emitter on the situation board and the match is verified, then the information contained in the observation is used to update the attributes of the matched emitter, including increasing the confidence level of the hypothesis that the emitter represents. Moreover, if the new observation is the second (or greater) observation of the emitter for the current time and it is from a different collection site than the previous observation(s) at that time, then a locational fix for the emitter is computed using the observed lines of bearing. If, in addition, the **Emitter-Type** and/or **Emitter-Mode** indicate a near-term threat to a friendly aircraft, then a threat report is output.

[4] Knowledge relating an aircraft type, for example F-15 or MIG-3, with the number and types of radars it carries is available. Using this knowledge and the identified emitter types in a cluster, it is possible to roughly estimate bounds on the number and types of aircraft in the cluster.

3. If the observation matches a known emitter but fails the match verification test, then an error in the **Emitter-Identifier** is indicated and the situation board is modified so as to undo any incorrect inferences based on the error. Also, an ide··ifier error report is output to the collection sites.

On a periodic basis, the status of each emitter on the situation board is evaluated and various actions are taken:

1. If there have been no recent observations of the emitter, then the confidence level of the emitter is reduced. If, as a consequence of this reduction, that level falls below a given *no-confidence* threshold, then the emitter and all of the consequences infered from it (including cluster association) are deleted from the situation board.

2. If the confidence level is above a given *full-confidence* threshold and the emitter is not currently associated with a known cluster, then an attempt is made to *match* the emitter with a cluster on the situation board. This match is based on the track and heading histories and the type attributes of the emitter and the cluster. If a match is made, then the emitter is associated with the matched cluster and the emitter's current attributes are used to update the attributes of the cluster. If the match fails, then a new cluster is hypothesized on the situation board and the emitter is associated with it.

3. In the remaining case of a recently observed emitter with an associated cluster, the current attributes of the emitter are used to update the attributes of its associated cluster.

Also on a periodic basis, the state of each hypothesized cluster on the situation board is examined. If all of the emitters associated with the cl ister have been deleted, then the cluster is deleted from the situation board. Otherwise:

1. The cluster is checked to see if it should be *split* into two (or more) clusters based on the currrent locations of its associated emitters. If so, new clusters with the appropriate associated emitters are hypothesized on the situation board.

2. The track history, heading history, speed history and activity history of the cluster are updated; and, if any new emitters have been recently associated with the cluster, an estimate of the types and numbers of aircraft comprising the cluster is derived.

3. A current status report for the cluster is output.

The ELINT processing flow lends itself naturally to concurrent execution. The parallel implementation of ELINT using CAOS is described in Section 4. The CAOS system itself is described in the following section.

3. The CAOS Programming Framework

CAOS is a framework which supports the encoding and the execution of multiprocessor expert systems. It represents an early attempt to bridge the gap between the application specification and the multiprocessor system programming primitives. The design of CAOS is predicated on the belief that many highly parallel architectures (e.g., hundreds of processors) will emphasize limited communication between processor-memory pairs rather than uniformly shared memory. We expect that such an architecture will favor relatively coarse-grained problem decomposition with little synchronization between processors. CAOS is intended for use in real-time, data interpretation applications such as continuous speech recognition and radar and sonar signal interpretation (see, for example, [9, 10]). CAOS is based on an object-oriented programming paradigm, and it draws many of its ideas from the Flavors system [1] and the Actors paradigm [11].

A CAOS application consists of a collection of communicating, active *agents*, each responding to a number of application-dependent, predeclared messages. An agent retains long-term local state. Each agent is a multi-process entity, that is, an arbitrary number of processes may be active at any one time in a single agent.[5] Conceptually, an agent can be thought of as virtual, multiprocess processor and memory pair. It responds to externally sent messages, and these message responses can alter the state of its local memory and can include the sending of messages to other agents.

CAOS is designed to express parallelism at a relatively coarse grain-size. For example, in the ELINT experiment, the message handlers (i.e., the *methods*) which implement the message responses are written as Lisp procedures, each averaging about one hundred lines of primitive Lisp code. CAOS supports no mechanism for finer-grained concurrency such as within the execution of agent processes, but neither does it rule it out. We could easily imagine message

[5]The active processes in an agent are not scheduled preemptively. Instead, an executing agent process either runs to completion or until it is "blocked" awaiting some remote service (see Section 5).

methods being written, for example, in QLisp [12], a concurrent dialect of CommonLisp which supports finer-grained concurrency.

3.1. CAOS' Approach to Concurrency

A CAOS application is structured to achieve high degrees of concurrency in the application execution in two principal manners: *pipelining* and *replication*. Pipelining is most appropriate for representing the flow of information between levels of abstraction in an interpretation system. Replication provides means by which the interpretation system can cope with arbitrarily high data rates.

3.1.1. Pipelining

Pipelining is a common means of parallelizing tasks through a decomposition into a linear sequence of concurrently operating stages. Each stage is assigned to a separate processing unit which receives the output from the previous stage and provides input to the next stage. Optimally, when the pipeline reaches a steady-state, each of the processors is busy performing its assigned stage of the overall task.

CAOS promotes the use of pipelines to partition an interpretation task into a sequence of interpretation stages where each stage of the interpretation is performed by a separate agent. As data enters one agent in the pipeline, it is processed, and the results are sent to the next agent. The data input to each successive stage represents a higher level of abstraction.

Sequential decomposition of a large task is frequently very natural. Structures as disparate as manufacturing assembly lines and the arithmetic processors of high-speed computing systems are frequently based on this paradigm.

Pipelining provides a mechanism whereby concurrency is obtained without duplication of mechanism (i.e., machinery, processing hardware, knowledge, etc.). In an optimal pipeline of n processing elements, the throughput of the pipeline is n times the throughput of a single processing element in the pipeline.

Unfortunately, it is often the case that a task cannot be decomposed into a simple linear sequence of subtasks. Some stage of the sequence may depend not only on the results of its immediate predecessor, but also on the results of more distant predecessors, or worse, some distant successor (e.g., in feedback loops). An equally disadvantageous decomposition is one in which some of the processing stages take substantially more time than others. The effect of either of these conditions is to cause the pipeline to be used less efficiently. Both these

conditions may cause some processing stages to be busier than others. In the worst case, some stages may be so busy that other stages receive almost no work at all. As a result, the n-element pipeline achieves less than an n-times increase in throughput. We discuss a partial remedy for this situation below.

3.1.2. Replication

Concurrency gained through replication is ideally orthogonal to concurrency gained through pipelining. Any size processing structure, from an individual processing element to an entire pipeline, is a candidate for replication. Consider a task which must be performed on the average in time t, and a processing structure which is able to perform the task in time T, where $T > t$. If this task were actually a single stage in a larger pipeline, this stage would then be a bottleneck in the throughput of the pipeline. However, if the single processing structure which performed the task were replaced by T/t copies of the same processing structure, the effective time to perform the task would approach t, as required. Replication is more costly than pipelining, but it does avoid some of the problems associated with developing a pipelined decomposition of a task.

Our work leads us to believe that such replicated computing structures are feasible, but not without drawbacks. Just as performance gains in pipelines are impacted by inter-stage dependencies, performance gains in replicated structures are impacted by inter-structure dependencies.

Consider a system composed of a number of copies of a single pipeline. Further, assume the actions of a particular stage in the pipeline affects each copy of itself in the other pipelines. In an expert system, for example, a number of independent pieces of evidence may cause the system to draw the same conclusion. The system designer may require that when a conclusion is arrived at independently by different means, some measure of confidence in the conclusion is increased accordingly. If the inference mechanism which produces these conclusions is realized as concurrently operating copies of a single inference engine, the individual inference engines will have to communicate between themselves to avoid producing multiple copies of the same conclusion rather than a composite conclusion. Any consistency requirement between copies of a processing structure decreases the throughput of the entire system, since a portion of the system's work is dedicated to inter-system communication. Examples of this situation are shown in Section 4 where we describe the CAOS agent types for the ELINT application.

3.2. Programming in CAOS

CAOS is basically a package of operators on top of Lisp. These operators are partitioned into three major classes -- those which declare agent classes, those which initialize agents, and those which support communication between agents. We now describe briefly the CAOS operators for each of these classes. A more complete description of these operators is given in [13].

3.2.1. Declaration of Agents

Agents classes, like most object-oriented classes, are declared within an inheritance network. Each agent class inherits the attributes of its (multiple) parents. The root CAOS agent class, **vanilla-agent**, contains the minimal attributes required of a functional CAOS agent. All other CAOS agents have the **vanilla-agent** as a parent, either directly or indirectly. Another CAOS-declared agent class, **process-agenda-agent**, is a specialization of **vanilla-agent**, and includes a priority mechanism for scheduling the execution of messages. The **vanilla-agent** schedules its messages in a FIFO manner only.

Application agent classes are declared by augmenting the following primary attributes of CAOS-declared or other ancestral agent classes:

Local-Variables: An instance agent's local variables store its private state. The agent's message handlers may refer freely to only those variables declared locally within the agent. Each local variable may be declared with an initial value.

Messages-Methods: The only messages to which an agent may respond are those declared in the agent's class declaration. Associated with each declared message name is the name of the message's *method* (i.e., the message's message handler). In CAOS, a method name must refer to a defined Lisp procedure. This declaration simplifies the task of a resource allocator which must load application code onto each CARE site.

Clocks-Methods: An agent may periodically invoke actions based on internal clock "ticks." For example, the periodic update of emitter agents and the periodic output of cluster status reports are invoked by clock ticks. A clock is defined by its tick interval. Whenever an internal agent clock ticks, the set of methods associated with that clock are scheduled for execution.

Critical-Methods: This attribute declares certain sets of methods as being mutually "critical

regions" for their owning agents.[6] Each such set of critical methods has an associated *lock*. Before an owning agent agent executes a critical method, this lock is checked. If it is unlocked, the agent locks it and executes the method. Upon completion of the method, the agent unlocks the lock. If the lock is locked, the method is queued in a FIFO queue awaiting the unlocking of the lock.

There are a number of additional basic agent attributes. However, most of these are used only internally by CAOS.

3.2.2. Initialization of agents

An initial CAOS configuration is specified by a two-component initialization form. The first component of the form creates the *static* agent instances. Some agent instances are created during system initialization and exist throughout a CAOS run. Such agent instances are called static agents as opposed to *dynamic* agents which are created (and possibly deleted) during program execution. For programmer convenience, we allow code in agent message handlers and default values of local-variables to reference such static agents by name. Before an agent instance begins running, each symbolic reference to the declared static agents is resolved by the CAOS runtimes.

The second component of the form is a list of expressions to be evaluated sequentially when CAOS's static agent instantiation phase is complete. Each expression is intended to send a message to one of the static agents declared in the first part of the form. These messages serve to initialize the application. For example, in the ELINT application the initialization messages open log files and start the processing of ELINT observations.

Agent instances may also be created dynamically during execution. The creation operator accepts an agent class name and a location specification.[7] The *remote-address* of the newly-created agent instance is returned. The remote-address of an agent includes the CARE site coordinates where the agent resides and a pointer to the agent in the address space of that

[6] A design goal for ELINT in CAOS was to avoid the use of critical methods, and our ELINT implementation does not use any. The CAOS initialization routines, however, do use such methods.

[7] Currently, agents may be created only "at" or "near" specified CARE sites. CAOS makes no attempt at dynamic load balancing.

site. A dynamically created agent may *not* be referenced symbolically, however, its remote-address may be exchanged freely.

3.2.3. Communications Between Agents

Agents communicate with each other by exchanging messages. CAOS does not guarantee when messages reach their destinations. Due to excessive message traffic or processing element failure, messages may be delayed indefinitely during routing. It is the responsibility of the application program to detect and recover from such delayed messages.

Two classes of messages are defined: those which return values, called *value-desired* messages, and those which do not, called *side-effect* messages. The value-desired messages are made to return their values to a special cell called a *future* which represents a "promise" for an eventual value.[8] Processes attempting to access the value of a future are blocked until that future has had its value set. Futures are first-class data types, and they may be manipulated by non-strict Lisp operators (e.g., list) even if they have not yet received a value. It is possible for the value of a CAOS future to be set more than once, and it is possible for there to be multiple processes awaiting a future's value to be set.

The CARE primitive **post-packet**, which sends a packet from one process to another, is employed in CAOS to produce three basic kinds of message sending operations:

post: The **post** operator sends a side-effect message to an agent. The sending process supplies a remote-address to the target agent (or its name in the case of a static agent), the message's routing priority, and the message's name and arguments. The sender continues executing while the message is delivered to the target agent.

post-future: The **post-future** operator sends a value-desired message to the target agent. The sending process supplies the same parameters as for **post**, and it is immediately returned a local pointer to the future which will eventually receive a value from the target agent. As for post, the sender continues executing while the message is being delivered and executed remotely. A process may later check the state of the future with the **future-satisfied?** operator or access the future's value with the **value-future** operator. This latter operator will *block* the process (i.e., suspend its execution and "swap it out") if the future has not yet received a value. When the

[8]Futures are also used in Multilisp [14]. The HEP Supercomputer [15] implemented a simple version of futures as a process synchronization mechanism.

future finally receives a value, the blocked process is rescheduled for resumed execution.

post-value: The **post-value** operator is similar to the **post-future** operator except that the sending process is immediately blocked until the target agent has returned a value. This operator is defined in terms of **post-future** and **value-future**, and it is provided for programming convenience.

It is possible to detect delay of value-desired messages by attaching a timeout to the associated future. The operators **post-clocked-future** and **post-clocked-value** are similar to their untimed counterparts but allow the caller to specify a *timeout-period* and *timeout-action* to be performed if the future is not set within the timeout-period. Typical timeout-actions include setting the future's value to a default value or resending the original message using the **repost** operator.

There also exist versions of the basic posting operators which allow the same message to be sent to multiple agents simultaneously. These versions exploit the multicast facilities of CARE (see Section 5).[9]

Multipost sends a side-effect message to a list of agents while **multipost-future** and **multipost-value** send value-desired messages to lists of agents. In the latter two cases, the associated future is actually a list of futures, and the future is not considered satisfied until all the target agents have responded. The value of such a message is an association-list where each entry in the list is composed of an agent's remote-address or name and the returned message value from that agent. There exist clocked versions of these operators (called, naturally, **multipost-clocked-future** and **multipost-clocked-value**) to aid in detecting delayed multicast messages.

3.3. The Runtime Structure of CAOS

CAOS is structured around three principal levels: site, agent, and process. Two of these levels, site and process, reflect the organization of CARE. The remaining agent level is an artifact of CAOS. We describe here only briefly the runtime structure of CAOS. This structure is described in greater detail in [13].

[9]Neither CAOS nor CARE currently support a "predicated multicast" mode wherein messages would be sent to all agents satisfying a particular predicate. Messages can only be multicast to a fully-specified list of agents. Receiving agents can, of course, apply arbitrary predicates to the message in order to determine their consequent action.

The implementation of CAOS described in this report is written in Zetalisp [1] and the primitive CARE operators using Zetalisp's object-oriented programming tool, Flavors[1].

Each CARE site contains a CAOS Site-Manager. A Site-Manager is realized as a Flavors instance. Its instance variables store site-global information needed by all agents located on the site. In addition, each Site-Manager includes CARE-level processes which perform the functions of creating new agents on its site and translating static agent symbolic names into agent addresses.

Each CAOS agent is also realized as a Flavors instance. A CAOS agent is a multiprocess entity. Most of the processes are created in the course of problem-solving activity. These processes are refered to as user processes. At runtime, however, there are always two special processes associated with each CAOS agent -- the *agent input monitor process* and the *agent scheduler process*. The agent input monitor process watches the CARE stream by which the agent is known to other agents. It handles request messages and responses from value-desired messages from these agents. CAOS user processes are created in response to request messages from other agents or clocked methods. The agent scheduler process collaborates with the CARE site's operator processor in the scheduling of these user processes (see Section 5).

4. ELINT's Implementation in CAOS

We describe now the agent types and their organization for the ELINT application as implemented in the CAOS framework. This implementation illustrates some of the benefits and some of the drawbacks of the framework. As discussed in Section 2, ELINT is an expert system whose domain is the interpretation of passively-observed radar emissions. ELINT is meant to operate in real time. Emitters appear and disappear during the lifetime of an ELINT run. The primary flow of information in ELINT as implemented in CAOS is through a pipeline with replicated stages. Each stage in the pipeline is an agent. The basic ELINT agent pipeline is illustrated in Figure 4-1

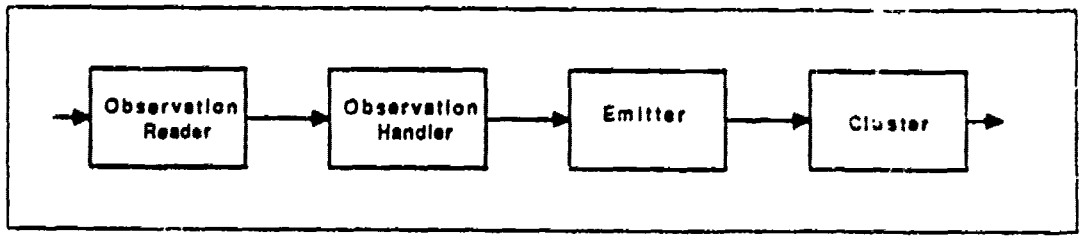


Figure 4-1: The basic ELINT agent processing pipeline.

4.1. ELINT Agent Types

The ELINT agent types described here are those used by the CT control strategy version of ELINT in CAOS (see Section 6).

Observation-Reader Agent

Observation-reader agents are an artifact of the simulated environment in which our ELINT implementation runs. Their purpose is to feed radar observations into the system. Observation-readers are driven off system clocks. At each clock "tick" (one ELINT time unit), they supply all observations for the associated time interval to the proper observation-handler agents. This behavior is similar to that of radar collection sites in an actual ELINT setting.

Observation-Handler Agent

The observation-handler agents accept radar observations from associated radar collection sites. Of course, in the simulated environment the observations actually come from observation-reader agents. There may be several observation-handlers associated with each collection site. The collection site chooses to which of its observation-handlers to pass an observation based on some scheduling criteria, for example, round-robin.

The contents of an ELINT observation was described in Section 2. In particular, each observation contains an identifier number assigned by the collection site to distinguish the source of the observation from other known sources. This source identifier is usually, but not always, correct. When an observation-handler receives an observation, it checks the observation's identifier to see if it already knows about the emitter which is the observation's source. If it does, it passes the observation to the appropriate emitter agent which represents the observation's source. If the observation-handler does not know about the emitter, it asks an emitter-manager agent to create a new emitter agent and then passes the observation to that new agent.

Emitter-Manager Agent

There may be many emitter-manager agents in the system. An emitter-manager's task is to respond to requests from observation-handlers to create new emitter agents with associated source identifier numbers. If there is no such emitter agent in existence when the request is received, the manager will create one and return its remote-address to the requesting

observation-handler agent. If there is such an emitter agent in existence when the request is received, the manager will simply return its remote-address to the requestor. This situation arises when one observation-handler requests an emitter that another observation-handler had previously requested. Emitter-managers must also handle the case of "almost concurrent" requests for the same emitter. This case occurs when a request is received for an emitter agent which is currently being created by another process on another CARE site in response to a slightly earlier request.

The reason for the emitter-manager's existence is to reduce the amount of inter-pipeline dependency with respect to the creation of emitters. When ELINT creates an emitter it is similar to a typical expert system drawing a conclusion based on some evidence. ELINT must create its emitters in such a way that the individual observation-handlers do not each end up creating copies of the "same" emitter, that is, creating multiple emitter agents with the same associated source identifier (see Section 3.1.2). Consider the following strategies that the observation-handler agents could use to create new emitter agents:

1. The handlers could create the emitter agents themselves immediately as needed. Since the collection sites may pass observations with the same source identifier to any observation-handler, it is possible for multiple observation-handlers to each create its own copy of the same emitter. This strategy is not acceptable.

2. The handlers could create the emitter agents themselves, but inform the other handlers that they have done this. This scheme breaks down when two handlers try simultaneously (or almost simultaneously) to create the same emitter.

3. The handlers could rely on a single emitter-manager agent to create all emitters. While this approach is safe from a consistency standpoint, it is likely to be impractical as the single emitter-manager could become a processing bottleneck.

4. The handlers could send requests to one of many emitter-managers chosen by some arbitrary method. This idea is nearly correct, but does not rule out the possibility of two emitter-managers each receiving creation requests for the same emitter.

5. The handlers could send requests to one of many emitter-managers chosen through some algorithm which is invariant with respect to the source identifiers.

This last strategy is the one used used in our implementation of ELINT. The algorithm for choosing which emitter-manager to use is based on a many-to-one mapping of source identifiers to emitter-managers.[10]

Emitter Agent

Emitter agents hold the state and history of the observation sources they represent. As each new observation is received by an emitter agent, it is added to a list of new observations. On a periodic basis, this list of new observations is scanned for interesting information. In particular, after enough observations are received, the emitter may be able to determine the heading, speed, and location of the source it represents. The first time it is able to determine this information, it asks a cluster-manager agent to either match the emitter to an existing cluster agent (as described in section 2.3) or create a new cluster agent to hold the single emitter. Subsequently, it sends an update message to the cluster agent to which it is associated indicating its current heading, speed, and location.

Emitters maintain a qualitative confidence level of their own existence (*possible*, *probable*, *positive* and *was-positive*). If new observations are received often enough, the emitter will increase its confidence level until it reaches *positive*. If an observation is not received by an emitter in the expected time interval, the emitter lowers its confidence by one step. If the confidence falls below *possible*, the emitter deletes itself, informing its manager and any cluster to which it is associated of its deletion.

Cluster-Manager Agent

The cluster-manager agents play much the same role in the creation of cluster agents as the emitter-manager agents play in the creation of emitter agents. However, it is not possible to compute an invariant to be used for a many-to-one mapping between emitters and cluster managers. If ELINT were to employ multiple cluster-managers, any strategy for which of the many managers an emitter agent chooses to request a cluster match could still result in the creation of multiple instances of the "same" cluster (i.e., multiple cluster agents representing the same physical cluster of emitters). Thus, we have chosen to implement ELINT using only a single cluster-manager. Fortunately, new cluster creation is a relatively rare event, and the

[10] The algorithm simply computes the source identifier modulo the number of emitter-managers and maps that number to a particular manager.

single cluster-manager has never been observed to be a processing bottleneck.

As described above, requests from emitters to associate themselves with clusters are specified as match requests over the extant clusters. Emitters are matched to clusters on the basis of their location, speed, and heading histories. However, the cluster-manager does not itself perform this matching operation. Although it knows about the existence of each cluster it has created, it does not know about the current state of those clusters. Thus, the cluster-manager asks all of its clusters to (concurrently) perform a match.

If none of the clusters responds with a positive match, the cluster-manager creates a new cluster for the emitter. If one cluster responds positively, the emitter is added to the cluster and it is so informed of this fact. If more than one cluster responds positively, this usually indicates that there is not yet sufficient resolution of the emitter's history to uniquely associate it with a cluster. In this case the emitter to cluster matching operation is tried again after more observations of the emitter have been processed.

Cluster Agent

The radar emissions from a cluster of emitters often indicate the activities of the aircraft represented by that cluster. For example, emissions from a missile guidance radar indicate that an air-to-air attack is imminent. Each cluster agent periodically applies heuristics about types of radar signals to try to determine the current activities of its represented aircraft, and, in particular, if these activities represent a threat to friendly aircraft. This activity information, the aircraft type information, and the merged track parameters of the emitters associated with each cluster are the primary outputs of the ELINT system. Also, each cluster periodically checks to see if all constituent emitters have been deleted. If so, it deletes itself.

Time-Manager Agent

Many of the knowledge-based actions taken by an ELINT agent make use of the agent's *last-observed* time, that is, the time stamp of the most recent observation associated directly or indirectly with the agent. For example, if an emitter agent determines that it has received no new associated observations for several data time intervals (i.e., that it is "out-of-date"), it will consider itself as no longer exisiting and it will delete itself and all of its relational links from ELINT's situation board.[11]

[11]This action reflects the expectation knowledge that if an emitter within the area of observation is observed at time t, then it is expected that it will be observed at time $t+1$.

In an asynchronous message passing system such as CARE, it is difficult for an agent to determine whether it is out-of-date because it has not been observed recently or because messages to it which would result in an update of its last-observed time are delayed due to overall system load or local load imbalances. One solution to this problem would be for each observation-handler agent to send an "end-of-observation-time-interval" message to each of its known emitter agents whenever it observes the crossing of an observation time interval boundary.[12]

This solution was rejected for the reported implementation of ELINT because of a perceived excessive message overhead.[13] Instead, our ELINT experiment uses a time-manager agent. Whenever an observation-handler agent observes a new input observation time stamp, it reports this new time to the time-manager via a message. The time-manager maintains a conservative, global current observation time which is the minimum of the the reported time stamps. Whenever any agent considers taking a drastic, non-reversible action which is based on its being out-of-date (e.g., deleting itself), it requests a confirmation from the time-manager that its (the requesting agent's) last-observed time is sufficiently older than the time-manager's global current observation time. The requesting agent does not perform its considered action until it receives the confirmation. If in the interim, the requesting agent receives any messages which result in an update of its last-observed time, the confirmation is ignored.

Reporter Agent

Instances of the reporter agent class are used to asynchronously output various ELINT reports to displays and/or files, for example, threat reports and periodic situation board reports. In addition, instances of a specialization of the reporter class, **debug-trace-reporter**, are used during application program debugging to asynchronously output debugging traces in a manner that minimally impacts system timing dependencies.

[12]Since each input observation stream is in observation-time sequential order, each observation-handler eventually knows when such a time boundary is crossed.

[13]This overhead may be more perceived than actual. A more recent implementation of ELINT uses such "end-of-observation-time-interval" messages. Initial results seem to indicate that the associated cost is not excessive (see [16]).

4.2. ELINT Agent Organization

The ELINT agents are basically organized as a pipeline with replicated stages where each stage is an agent. Inter-pipeline dependencies and dependencies between replicated stages are managed by emitter-manager and cluster-manager agents. The amount of replication (i.e., the number of agents) at each pipeline stage is a function of that stage. For some stages, the number of replicated agents at that stage is fixed during system initialization. For example, the numbers of observation-handler agents, emitter-manager agents, and cluster-manager agents are pre-determined based on the number of collection sites and their output data rates. The numbers of emitter stages and cluster stages vary during the course of execution since the corresponding emitter agents and cluster agents are created and deleted as the radar emitters and collections of radar emitters which they represent appear and disappear over time.

The overall organization of the ELINT agents is illustrated in Figure 4-2

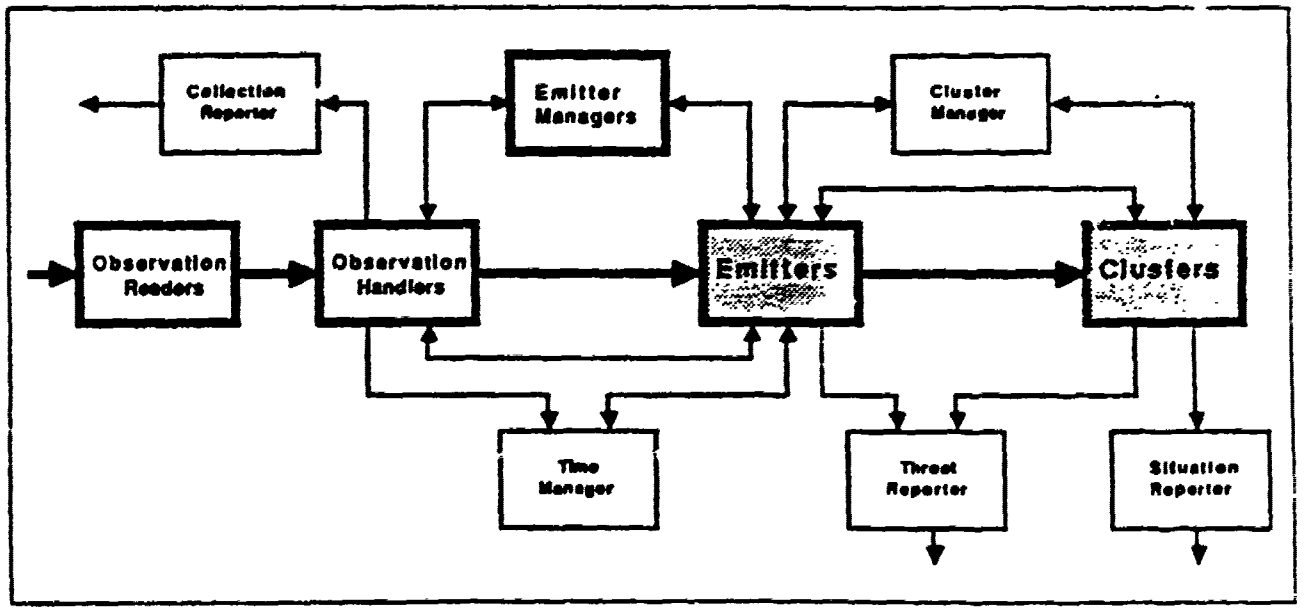


Figure 4-2: The overall ELINT agent communication organization.

5. An Overview of CARE

The CARE architectural specification and its simulation environment provide a parameterized and instrumented multiprocessor simulation testbed designed to aid research in alternative parallel architectures. The testbed executes within SIMPLE, a hierarchical, event-driven simulator [3].

A CARE architecture is a grid of tens to hundreds of processing sites interconnected via a

dedicated communications network. The network uses dynamic, buffered, cut-through routing, and it supports multicast inter-site message transmission. The ELINT experiment, for example, was performed on various square CARE grids of hexagonally connected sites, that is, each site, excluding those at the edges of the grid, is connected to six of its eight nearest neighbors.

As shown in Figure 5-1, each CARE site consists of an *evaluator*, a general-purpose processor-memory pair; an *operator*, a dedicated communications and process scheduling processor which shares memory with the evaluator; and network interfaces -- *net-inputs* and *net-outputs* -- that accomplish pipelined message transmission, flow control, deadlock avoidance, and routing. Each net-input at a site may establish a connection with a net-output at any site, and all such connections at a site may be simultaneously active.

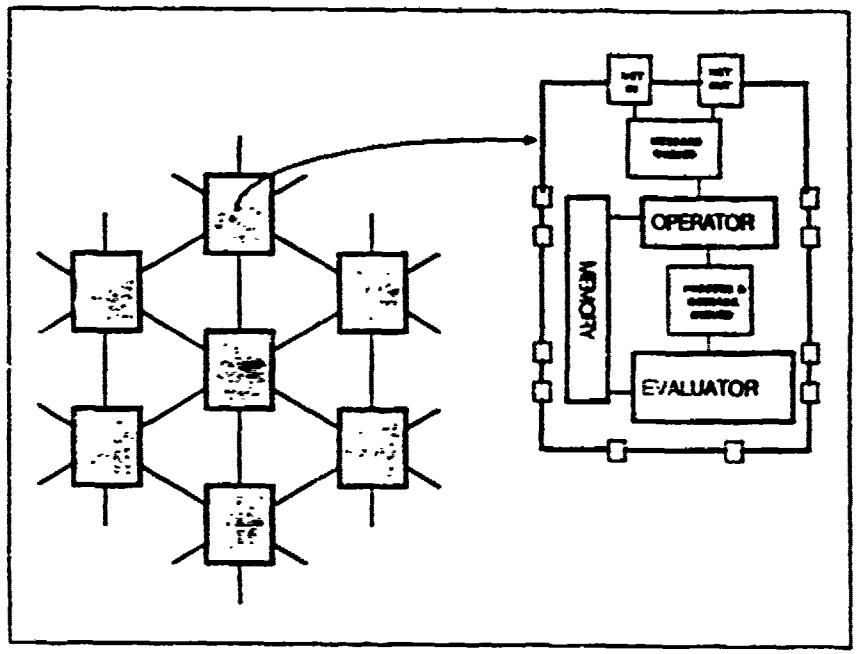


Figure 5-1: A hexagonally connected CARE grid.

Application-level computations take place in the evaluator. The operator performs two duties. As a communications processor, it is responsible for initiating and receiving messages. As a scheduling processor, it queues application-level processes for execution in the evaluator. Message routing is performed by the net-input and net-output network interfaces.

In our simulation of CARE, the evaluator is treated as a "black box" Lisp processor. None of its internal operation is simulated. The Lisp machine hosting the simulation serves as the evaluator in each processing site. The operator, however, is functionally simulated, and the network interfaces are simulated and instrumented in great detail.

CARE allows a number of parameters of the processor grid to be adjusted. Among these parameters are: the speed of the evaluator, the speed of the communications network, the network routing algorithm, and the speeds of the process creating and switching mechanisms. By altering these parameters, a single processor grid specification can be made to simulate a wide variety of actual multiprocessor architectures. For example, we can experiment with the optimal level-of-granularity of problem decomposition by varying the speed of both process-switching and communications. Alternative network topologies can be studied by using SIMPLE's graphic interfaces and composition operators to configure CARE components into any topology that can be wired.

The CARE simulation environment provides detailed displays of such information as evaluator, operator, and communication network utilization, and process scheduling latencies. This instrumentation package informs developers of CARE applications of how efficiently their systems make use of the simulated hardware.

A more detailed description of CARE is given in [16], and the technology considerations underlying the CARE architecture are discussed in Appendix I.

6. Results and Conclusions

The CARE architectural simulation testbed and the CAOS system we have described have been fully implemented, and they are in use by several groups within our Architectures Project. CAOS-CARE executes on the Symbolics 3600 family of machines as well as on the Texas Instruments Explorer Lisp machine. ELINT, as described in Sections 2 and 4, has also been fully implemented, and we have analyzed its performance on various size CARE grids.

6.1. Evaluating CAOS

CAOS is a rather special-purpose environment, and it should be evaluated with respect to the programming of concurrent, real-time signal interpretation systems. In this section, we explore CAOS's suitability along the dimensions of expressiveness, efficiency, and scalability.

6.1.1. Expressiveness

When we ask that a language be suitably expressive, we ask that its primitives be a good match to the concepts the programmer is trying to encode. The programmer should not need to resort to low-level "hackery" to implement operations which ought to be part of the language. We believe we have succeeding in meeting this goal for CAOS (although to date, only CAOS's designers have written CAOS applications). Programming in CAOS is essentially programming

in Lisp using objects but with added features for declaring, initializing, and controlling concurrent, real-time signal interpretation applications.

6.1.2. Efficiency

CAOS has a very complicated architecture. The lifetime of a message involves numerous processing states and scheduler interventions. Much of this complexity derives from the desire to support alternate scheduling policies within an agent. The cost of this complexity is approximately one order of magnitude in processing latency. For the common settings of simulation parameters, CARE messages are exchanged in about 2 to 3 milliseconds, while CAOS messages require about 30 milliseconds. It is this cost which forces us to decompose applications coarsely, since more fine-grained decompositions would inevitably require more message traffic.

We conclude that CAOS does not make efficient use of the underlying CARE architecture. This conclusion has lead to an evolution of both CAOS and CARE which is described briefly in Section 6.3 and in detail in [16].

6.1.3. Scalability

A system which scales well is one whose performance increases commensurately with its size. Scalability is a common metric by which multiprocessor hardware architectures are judged. For example, does a 100-processor realization of a particular architecture perform ten times better than a 10-processor realization of the same architecture? Does it perform only five times better, only just as well, or does it perform even worse? In hardware systems, scalability is typically limited by various forms of contention in memories, busses, etc. The 100-processor system might be no faster than the 10-processor system because all interprocessor communications are routed through an element which is only fast enough to support ten processors.

We ask the same question of a CAOS application. Does the throughput of ELINT, for example, increase as we make more processors available to it? This question is critical for CAOS-based, real-time interpretation systems. Our only means of coping with arbitrarily high data rates is by increasing the number of processors.

We believe CAOS scales well with respect to the number of available processors. The potential limiting factors to its scaling are increased software contention, such as the inter-pipeline bottlenecks described in Section 3, and increased hardware contention, such as overloaded processors and/or communication channels. Software contention can be minimized by the

design of the application. Communications contention can be minimized by executing CAOS on ... of an appropriate hardware architecture such as that afforded by CARE. CAOS applications tend to be ... y decomposed. They are bounded by computation, rather than communication, and communications loading was not a problem in our ELINT-CAOS-CARE experiment.

Unfortunately, processor loading remains an issue. A configuration with poor load balancing in which some CARE sites are busy while others are idle does not scale well. Increased throughput is limited by contention for processing resources on overloaded sites while resources on unloaded sites go unused. The problem of automatic load balancing is not addressed by CAOS as agents are simply assigned to processing sites on a round-robin basis with no attempt to keep potentially busy agents apart. We currently have no solution to the problem of processor load balancing beyond that of carefully "hand crafting" a site allocation strategy for each application and then "tuning" that strategy via succesive refinement.

6.2. Evaluating ELINT Under CAOS

The input data set used for most of our ELINT-CAOS runs was based on a scenario involving 16 aircraft mounting a total of 88 radar emitters with between 4 and 45 emitters active and observed during any one data time interval. The scenario takes place in a 60 by 80 mile area over 36 time units, and it involves 1040 separate emitter observations.

Our experience with ELINT indicates that the primary determiner of throughput and solution quality is the strategy used in making individual agents cooperate in producing the desired interpretation. Of secondary importance is the degree to which processing load is evenly balanced over the processor grid. We now discuss the impact of these factors on ELINT's performance.

The following three "control" strategies were used in our experiment:

1. NC: This "no control" strategy represents limited inter-agent control. Agents initiate actions independently. Whenever an agent wants to perform an action, it does so as soon as processing resources are available. For example, whenever an observation-handler agent needs a new emitter agent, it simply creates it with no attempt to coordinate this creation with other observation-handlers. As a result, multiple, non-communicating copies of an emitter may be created, and each copy receives a only portion of the input data it requires. The NC strategy was expected to produce qualitatively poor results, and it was primarilly intended only as a

baseline against which to compare more realistic control strategies. What was surprising was that the strategy also produced quantitatively poor results (see below).

2. CC: In this strategy, agents cooperate in the creation of new agents via manager agents as described in Section 4. The manager agents assure that only one copy of an agent is created, irrespective of the number of simultaneous creation requests. All requestors are returned a reference to the single new agent. Originally, we believed the CC (for "creation control") strategy would be sufficient for ELINT to produce satisficing high-level interpretations. Our experiment results showed that this was not always the case (see below).

3. CT: The CT ("creation and time control") strategy was designed to additonally manage the skewed views of real-world time which develop in agent pipelines. For example, this strategy prevents an emitter agent from deleting itself when it has not received a new observation in a while even though some observation-handler agent has sent the emitter an observation which it has yet to receive. The agents corresponding to the CT strategy are those described in Section 4.

Table 6-1 illustrates the qualitative effects of the various control strategies and grid sizes. The table presents the six major performance attributes by which the quality of an ELINT run is measured. Since the input data for the ELINT experiment were generated from known scenarios, it was possible to compare the results of an ELINT run with "ground truth."

Table 6-1: ELINT Solution Quality Versus Control Strategies and Grid Sizes.

Qualitative performance attribute	Control strategy/grid size					
	NC/16	CC/16	CC/36	CT/4	CT/16	CT/36
False alarms	1%	0	0	0	0	0
Reincarnation	49%	42	2	0	0	0
Confidences	19%	20	90	89	93	95
Fixes	48%	42	99	100	100	100
Threats	65%	63	81	87	87	90
Fusion	0%	0	77	85	88	89

The major qualitative performance attributes are:

False Alarms: This attribute is the percentage of emitter agents that ELINT should not have

hypothesized as existing with respect to the total number of emitter agents hypothesized.

ELINT was not severely impacted by false alarms in any of the control configurations in which it was run as the knowledge used for hypothesizing new emitters was quite conservative. That is, the knowlege was such that it prefered missing a true, but low confidence, emitter to creating a false alarm emitter.

Reincarnation: This attribute is the percentage of recreated emitter agents, that is, emitters which had previously existed but had erroneously deleted themselves due to lack of recent observations, with respect to the total number of emitters created. Large numbers of reincarnated emitters indicate some portion of ELINT is unable to keep up with the data rate. This can be caused by the data rate being too high globally so that all emitters are overloaded or by the data rate being too high locally due to poor load balancing so that some subset of the emitters are overloaded.

The CT control strategy was designed to prevent reincarnations. Hence, none occurred when CT was employed on any size grid. When the CC strategy was used, only the 36 site grid was large enough for ELINT to sufficently keep up with the input data rate so that emitters were not erroneously deleted due to overloa

Confidence Level: This attribute is the p[illegible]age of correctly deduced confidence levels for the existence of an emitter with respect to the [illegible]l number of times such confidence levels were determined.

For each hypothesized emitter, ELINT maintains a dynamic confidence level for the existence of the emitter based on accumulating evidence (see Section 4.1). The correct calculation of confidence levels depends heavily on the system being able to cope with the incoming data rate. One way to improve confidence levels was to use a large processor grid. The other was to employ the CT control strategy.

Fixes: This attribute is the percentage of correctly-calculated positional fixes of emitters with respect to the total number of times fixes could have been determined from the ground truth data.

A fix can be computed whenever an emitter has seen at least two observations from different collection sites in the same data time interval. If, for example, an emitter is undergoing reincarnation, it will not accumulate enough data to regularly compute fixes. Thus, the approaches which minimized reincarnation tended to maximize the correct calculation of fix

information.

Threats: As described in Sections 2 and 4, certain emitter and cluster events represent immediate threats. This attribute is the percentage of recognized threats with respect to the total number of threat events based on the ground truth data.

Fusion: This attribute is the percentage of correct clustering of emitter agents to cluster agents. The correct computation of fusion appeared to be related, in part, to the correct computation of confidence levels. The fusion process is also the most knowledge-intensive computation in ELINT, and our imperfect results indicate the extent to which ELINT's knowledge is incomplete.

The overall goal of the control strategy experiments was to see if it was possible to determine strategies where the quality of the output results were relatively insensitive to grid size and load balance but still achived significant concurrency.

We interpret from Table 6-1 that the control strategy has the greatest impact on the quality of results. The CT strategy produced high-quality results irrespective of the number of processors used. The CC strategy, which is much more sensitive to processing delays, performed nearly as well only on the 36 site grid. We believe the added complexity of the CT strategy, while never detrimental, is primarily beneficial when the interpretation system might be overloaded by high data rates or poor load balancing.

Table 6-2 gives the simulated execution times for the ELINT runs used to derive the data in Table 6-1, and Table 6-3 gives the total CAOS message counts for these runs.

Table 6-2: Simulated ELINT execution times for various control strategies and grid sizes.

Control strategy	Grid size		
	4	16	36
NC		>11.19 sec.	
CC		10.87	5.12
CT	11.80	8.10	4.17

Tables 6-2 and 6-3 clearly show that the processing cost of added control is far outweighed by the benefits in its use. Far less message traffic is generated, and the overall simulated time is reduced. Note that for the runs whose execution times are shown in Table 6-2, the input data

Table 6-3: CAOS message counts for ELINT executions with various control strategies and grid sizes.

Control strategy	Grid size		
	4	16	36
NC		>16118 msg.	
CC		7375	4823
CT	4516	4703	4616

rate was .1 seconds per ELINT time unit. Since the input data set used for these runs spanned 36 time units, the last observation was fed into the system at 3.6 (simulated) seconds. Hence, this is the minimum possible simulated execution time for these runs.

Table 6-4 and Figure 6-1 show the quantitative effect of processor grid size when the CT control strategy is employed. These results were produced with the input data rate set ten times higher (.01 seconds per ELINT time unit) than that used to produce Table 6-2. The minimum possible simulated execution time for the runs used to produce Table 6-4 is 0.36 seconds.

Table 6-4: Simulated ELINT execution time versus grid size for production runs using CT control strategy.

Grid size	Execution time
1	9.476 sec.
4	3.237
9	1.517
16	.761
25	.541
36	.557

As shown in Figure 6-1, the speedup achieved by increasing the processor grid size is nearly linear in the 1 to 25 processor site range. However, the 36 site grid was slightly slower than

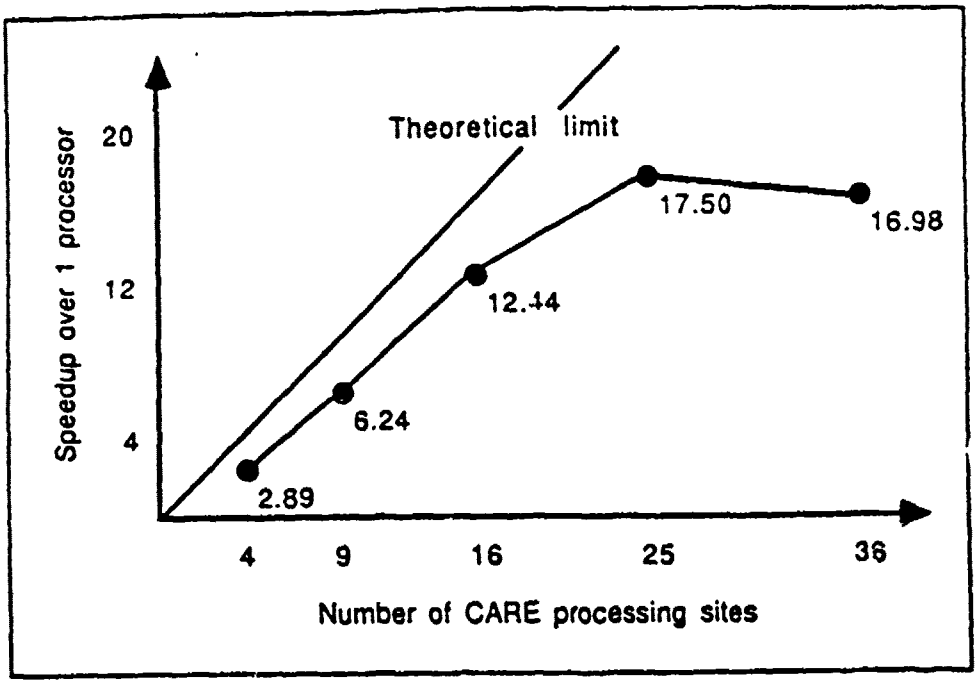


Figure 6-1: The relative speedup of ELINT executions on various size CARE grids.

the 25 site grid.[14]

In this last case, there was not sufficient data per ELINT time interval to warrant the additional processors. That is, there was not enough concurrency to exploit 36 processors. This can be seen from Table 6-5 which gives timing results for larger data sets with more emitters and observations during each time interval and, hence, more potential for concurrency.

Table 6-5: Simulated ELINT execution times and speedup for larger data sets.

Number of Observations	1-site grid execution time	36-site grid execution time	Speedup of 36 over 1
1040	9.476 sec.	.557 sec.	17.0
2080	25.10	.948	26.5
4160	55.87	2.259	24.7

As shown in this table, for an input data set representing twice as many emitters and

[14]Because of the intrinsic non-determinism of a CARE architecture, we observed variations in the solution qualities and the run times between different runs of the same input data set on the same size CARE grids. For such runs, the variations in solution qualities never exceeded a fraction of a percent. However, the varitions in run times where as much as five percent. This accounts for the slightly longer execution time on 36 versus 25 processors.

observations than the basic data set, the 36 site grid achived a speedup factor of 26.5 (as opposed to a speedup of 17.0 for the basic data set) over a single processor. However, for a data set four times larger than the basic data set, the speedup factor was only 24.8. This was because this larger, and hence more concurrent, data set saturated the 36 site grid. That is, the 2080 observation data set already provided enough concurrency to fully exploit the 36 site grid.

6.3. Some Open Questions

CAOS has been a suitable framework in which to construct concurrent signal interpretation systems, and we expect many of its concepts to be useful in our future computing architectures. Of principal concern to us now is increasing the efficiency with which the underlying CARE architecture is used. In addition, our experience suggests a number of questions to be explored in future research:

- What is the appropriate level of granularity at which to decompose problems for CARE-like architectures?

- What is the most efficient means to synchronize the actions of concurrent problem solvers when necessary?

- How can flexible scheduling policies be implemented without significant loss of efficiency? What is the impact on problem solving if alternate scheduling policies are not provided?

- Are there efficient mechanisms for dynamically balancing processor loads?

We have started to investigate these questions in the context of a new CARE environment. One of the primary difference between the original environment and the new environment is that the process is no longer the basic unit of computation. While the new CARE system still supports the use of processes, it emphasizes the use of *contexts* which are computations with less state than those of processes.

When a context is forced to suspend to await a value from a remote service, it is aborted, and restarted from scratch later when the value is available. This behavior encourages more fine-grained decomposition of problems written in a functional style where individual methods are small and consist of a binding phase followed by an evaluation phase.

In addition, CARE now supports arbitrary prioritization of messages delivered to streams. As

a result, it is no longer necessary to include in CAOS a complex and expensive scheduling strategy. Early indications are that the new CARE environment with a slightly modified CAOS environment performs around two orders of magnitude faster than the configuration described in this paper. The evolution of CARE and CAOS based on the results of our ELINT-CAOS -CARE experiment is described in greater detail in [16].

Acknowledgements

Our thanks to Russell Nakano, Sayuri Nishimura, [illegible] Nakul Saraiya who helped implement and maintain the CARE environment. Also, we wish to thank the staff of the Symbolic Systems Resources Group of the Knowledge Systems Laboratory for their excellent support of our computing environment. We express special gratitude to Edward Feigenbaum. His continued leadership and support of the Knowledge Systems Laboratory and the Architectures Project made it possible us to do the reported research.

I. Technology Considerations Underlying the CARE Architecture

The CARE simulation testbed can be used to simulate shared memory as well as message passing multiprocessor architectures. For example, it has been configured to simulate a single address space, shared global memory architecture where the processors (and their local cache memories) are connected to the shared memory's controllers via a switching network. However, the intended focus of the CARE testbed is on message passing, multiprocessor architectures where each processor has significant local memory. This focus is based on technology considerations -- primarily communication versus processing costs.

The base for development of general purpose multiprocessor systems, as for computer systems generally, is given by the design constraints and opportunities established by evolving semiconductor design and manufacturing processes. The VLSI design medium brings a new perspective on cost -- switches are cheap while wires are expensive. Communication costs dominate those associated with logic. Communication is currently the resource in shortest supply, and it will become more of a constraint rather than less as semiconductor lithographies decrease.

The consequence of relatively expensive communication is that performance is enhanced if the design establishes that whenever a lot of information has to move in a short time, it does not have to move far. Significant locality of high bandwidth links is a design goal. Among the highest bandwidth links in a computer system are those connecting the processor and memory. Thus, close coupling of processors with local memory is preferred.

To reduce demand on the communications resource to supportable levels, local memory sizes for multiprocessors can be expected to increase to the 100K byte level and beyond, and block transfers between backing store and such several hundred kilobyte local memories will be used to make the most efficient use of both memory structures and communications facilities. Moreover, the functionallity of memory controlers will expand to include, for example, management of request queues, the dispatching of results, and execution of synchronization primitives; and thus, the distinctions between a memory controller and a small, simple processor will become blurred.

The proportion of area for a simple, high performance processor to the total area of a site with, for example, 256K bytes of local storage can be reasonably estimated at around 15%. From (i) this estimate of the incremental cost of adding a processor to a block of memory, (ii) the significant size of the total local storage in the system, (iii) the blurring of distinctions

between fast, simple processors and memory controllers of increasing complexity, and (iv) the tendency towards block tranfers between local memory and backing store, it follows that the level of the storage hierarchy now labeled as "random access memory" is likely to be subsumed by a combination of large local memories and fast, block access backing stores in multiprocessor systems.

The performance of the available communication resource merits special attention in the design of multiprocessor systems. For example, dynamic routing which selects available inter-site links as needed is useful in balancing load, and thus it allows more of the communication resource of the system to be exploited throughout a computation. Cut-though routing which makes a routing decision on the fly as a packet is received reduces buffer requirements in the system and minimizes latency experienced in network transit. Flow control via signalling transmission delays back to the source based on local blockage information together with single "word" buffering and transmission validation at each network input and output port allows the source to complete a transmission in a time that does not depend on the size of the network. Point to point multicast which sends (approximately) the same packet to multiple targets using common resources to the largest degree possible can significantly enhance overall communication performance. A communication resource with these features provides a multiprocessor system with "virtual busses" that are established precisely as and when they are needed.

These technology considerations have led us to focus our attention on the class of multiprocessor hardware system architectures exemplified by CARE.

References

1. Weinreb, D. and Moon, D. (1981) Lisp machine manual, 4th ed. Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

2. Delagi, B., et al. (1986) CARE user's manual. Technical Report, Knowledge Systems Laboratory, Stanford University.

3. Saraiya, N. (1986) Simple user's manual. Technical Report, Knowledge Systems Laboratory, Stanford University.

4. Saraiya, N. (1986) AIDE: A distributed environment for design and simulation. Technical Report, Knowledge Systems Laboratory, Stanford University.

5. Williams, M., Brown, H. and Barnes, T. (1984) TRICERO design description. Technical Report ESL-NS539, ESL, Inc.

6. Aiello, N., Bock, C., Nii, H. P. and White, W. (1981) Joy of AGEing. Technical Report, Heuristic Programming Project, Stanford University.

7. Nii, H. P. (1986) Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. AI Magazine, vol. 7, no. 2, pages 38-53.

8. Nii, H. P. (1986) Blackboard systems part two: Blackboard application systems. AI Magazine, vol. 7, no. 3, pages 82-106.

9. Erman, L., Hayes-Roth, F., Lesser, V. and Reddy, D. R., (1980) The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. ACM Computing Surveys, vol. 12, pages 213-253.

10. Nii, H. P., Feigenbaum, E., Anton, J. and Rockmore. A. (1982) Signal-to-symbol transformation: HASP/SIAP case study. AI Magazine, vol. 3, no. 3, pages 23-35.

11. Lieberman, H. (1981) A preview of Act1. Artificial Intelligence Laboratory Memo 625., Massachusetts Institute of Technology.

12. Gabriel, R. and McCarthy, J. (1984) Queue-based multiprocessing Lisp. In Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming. Austin, Texas.

13. Schoen,E. (1986) The CAOS system. Technical Report, Knowledge Systems Laboratory, Stanford University.

14. Halstead, R. H., Jr. (1984) MultiLisp: Lisp on a multiprocessor. In Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming. Austin, Texas.

15. Denelcor, Inc. (1981) Heterogeneous element processor: Principles of operation. Boulder, Colorado.

16. Delagi, B., et al. (1986) Lamina: Streams and objects for concurrency. Technical Report, Knowledge Systems Laboratory, Stanford University.

Knowledge Systems Laboratory
Report No. KSL-87-02

January 1987

A Point-to-Point Multicast Communications Protocol

Gregory T. Byrd†
Department of Electrical Engineering
Stanford University
Stanford, CA 94305

Russell Nakano
Department of Computer Science
Stanford University
Stanford, CA 94305

Bruce A. Delagi
Worksystems Engineering Group
Digital Equipment Corporation
Maynard, MA 01754

This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266575.

†G. Byrd is supported by an NSF Graduate Fellowship, with additional support provided by the EE Dept.

Abstract

Many network topologies have been proposed for connecting a large number of processor-memory pairs in a high-performance multiprocessor system. In terms of performance, however, the communications protocol decisions may be as crucial as topology. This paper describes a protocol to support point-to-point interprocessor communications with multicast. Dynamic, cut-through routing with local flow control is used to provide a high-throughput, low-latency communications path between processors. In addition, multicast transmissions are available, in which copies of a packet are sent to multiple destinations using common resources as much as possible. Special packet terminators and selective buffering are introduced to avoid deadlock during multicasts. A simulated implementation of the protocol is also described.

1 Introduction

Many network topologies have been proposed for connecting a large number of processor-memory pairs in a high-performance multiprocessor system [1]. These topologies are often evaluated in terms of the average number of hops traversed by a packet, for example. However, the network performance may depend as much on its communication protocol as on its physical topology. For example, suppose the average number of hops in a network is M and the average packet length is N. In a store-and-forward network, the transmission time of a packet would be proportional to $M \times N$. If cut-through switching is used, however, the transmission time would be proportional to $M + N$, a significant difference for relatively large values of M or N. An appropriate communications protocol, then, is crucial if the full benefits of a topology are to be realized.

The protocol described in this paper is designed to fully utilize network resources. Dynamic, cut-through routing with local flow control is used to provide a high-throughput, low-latency communications path between processors. In addition, a multicast facility is provided, in which copies of a packet are sent to multiple destinations, using common resources as much as possible.

Dynamic routing means that the communications channel to be used is chosen at transmission time, based on what channels are available. The alternative, static routing, would prescribe a specific channel for every destination—if that channel were not available, the transmission would be blocked. Dynamic routing, by adapting to current channel usage, attempts to balance the network load. It is especially useful when the communications traffic is unpredictable or variable over time [2]. Balancing the load allows more of the communications resources of the system to be well used throughout a computation.

Cut-through routing [3] means that a routing decision is made on the fly, as a packet is received, rather than first buffering the entire packet and then deciding what to do with it.[1] This reduces buffering requirements in the system, since the packet does not need to be stored at intermediate points in the transmission [2] Kernami and Kleinrock [5] demonstrate that the cut-through approach outperforms both circuit switching and message switching (store-and-forward) when the communication paths are short, network utilization is relatively high, and messages are fairly small.

Flow control, in general, is any mechanism which attempts to regulate the flow of information from a sender to match the rate at which the receiver can accept it [8]. In this protocol, a transmission may be blocked and resumed in the event of network congestion. If an output channel becomes blocked, the sender stops sending data and halts the flow of data from upstream. When the channel becomes unblocked, the transmission is continued from where it was

[1] A related concept is staged circuit switching, described in [4]

[2] Cut-through switching as described in [3] requires that the packet be completely buffered if the output channel is blocked. In this protocol, no further data will be received from downstream until the channel becomes available Thus, packet buffering is not required

halted. The flow control mechanism is local, because actions are taken based on the state of the downstream component rather than global information about the entire network.

Multicast transmissions in a point-to-point network allow a packet to be sent to multiple destinations, using common resources as much as possible. The packet is replicated as needed, and subsets of the original target list are assigned to the copies. Thus, "virtual busses" are available precisely as and when they are needed. Selective buffering and special packet terminators allow potential deadlock conditions in multicasts to be detected and avoided.

The network components which define the protocol are introduced in Section 2, and the protocol itself is described in Sections 3 and 4. Finally, Section 5 describes an implementation of the protocol in the CARE simulation system.

2 Components

This section defines the network components used by the protocol. The protocol is defined by the behavior of these components and the values that are passed among them. Of course, these components do not necessarily correspond to distinct physical entities in a machine which implements this protocol—they are merely a useful means of specifying the functional behavior of such a machine.

The *site* component corresponds to a processor-memory pair in the target machine. In particular, a site contains an operator, an evaluator, a router, some local storage, and some network interface components, which are called net-inputs and net-outputs (see Figure 1).

The *evaluator* is the part of the site which executes application code. The evaluator can request network activity, but otherwise has no role in the network behavior of the system, so very little will be said about it in this paper.

The *operator* is responsible for handling system-level activity, including communication. In the target machine, it would create packets to be sent over the network and accept transmissions destined for its associated processor. The operator and evaluator communicate through shared local memory. The details of this communication will not be addressed in this paper.

The site components which interface directly to the network are called *net-inputs* and *net-outputs*. On each site, there is a net-input/net-output pair connected to the operator, for local packet origination and delivery, as well as a pair for every communication channel to the network.[3] We will refer to the pair connected to the operator as the "local" net-input and net-output.

The net-input is responsible for accepting a packet, making connections (using the router) to one or more net-outputs, and sending it on its way. The net-output is concerned with delivering the packet to a particular location, either the local operator or the next site on the transmission path. Note that,

[3] The exact number of net-inputs/net-output pairs required by a site depends on the network topology.

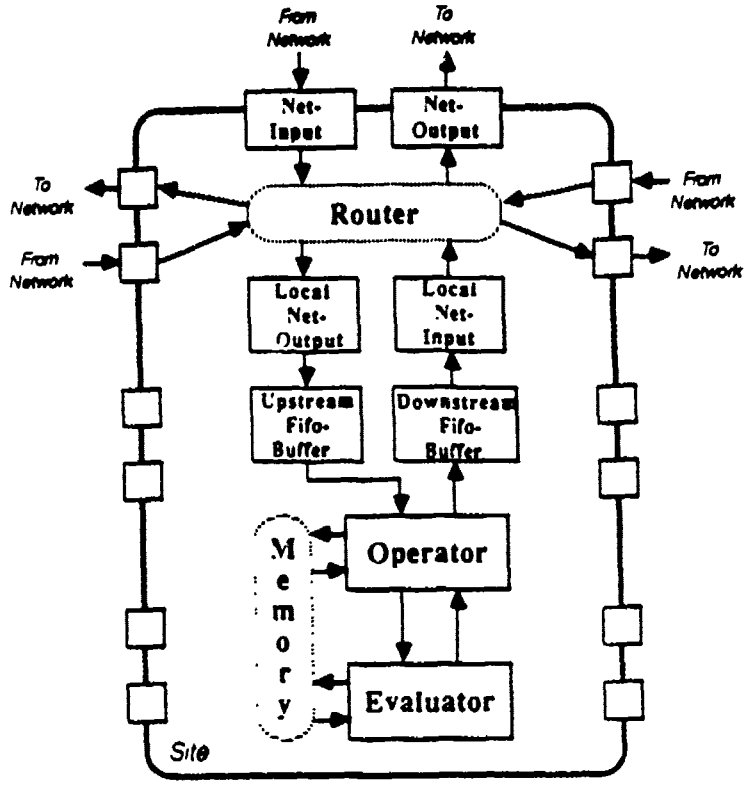


Figure 1: Components of a CARE *site*.

because of cut-through routing, net-inputs and net-outputs are only required to have enough storage for one word of a packet, rather than the entire packet.

The *router* connects all the net-inputs on a site to all the net-outputs. When it receives a packet from a net-input, it determines the destination (or destinations) and makes the connection to the appropriate net-output (or net-outputs). Also, flow control information from the net-outputs are relayed by the router to the appropriate net-input.

A pair of buffers, called *fifo-buffers*, queue packets between the operator and local net-input and net-output. The *upstream* fifo-buffer queues packets from the network to the operator; the *downstream* queues packets from the operator to the network.

3 Protocol Overview

3.1 Packets

Figure 2 shows the organization of a packet. The first part a packet is devoted to the *target entries*. Each entry contains a target address, a pointer to data within the packet, and flags indicating the last target in the list.

Following the target addresses are zero or more words of *data* and a one-word *packet terminator*. There are several distinct packet terminators, as shown in Table A, which are used by the operator to determine the status of the packet.[4]

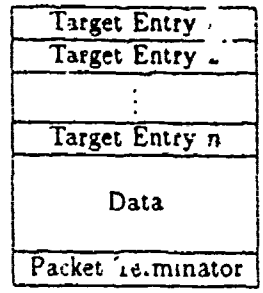


Figure 2: Organization of a packet

Terminator	*Meaning*
:end-of-packet	Normal packet termination.
:abort-packet	Packet is to be discarded by operator
:local-end-of-packet	Treat as **:end-of-packet**, except ignore all packet targets other than the local site.

Table A: Packet terminators.

3.2 Packet Transmission

The transmission path of a packet is shown in Figure 3. First, an evaluator requests a packet transmission. The operator then sends the packet (through a buffer) to the local net-input. For the moment, assume that there is only one target for the packet. (This is called a *unicast* transmission.) The router then decides which net-output should receive the packet, based on the target address and the availability of net-outputs, sets up a connection between the local net-input and the selected net-output, and begins the transfer of the

[4] As described in Subsection 4.3.

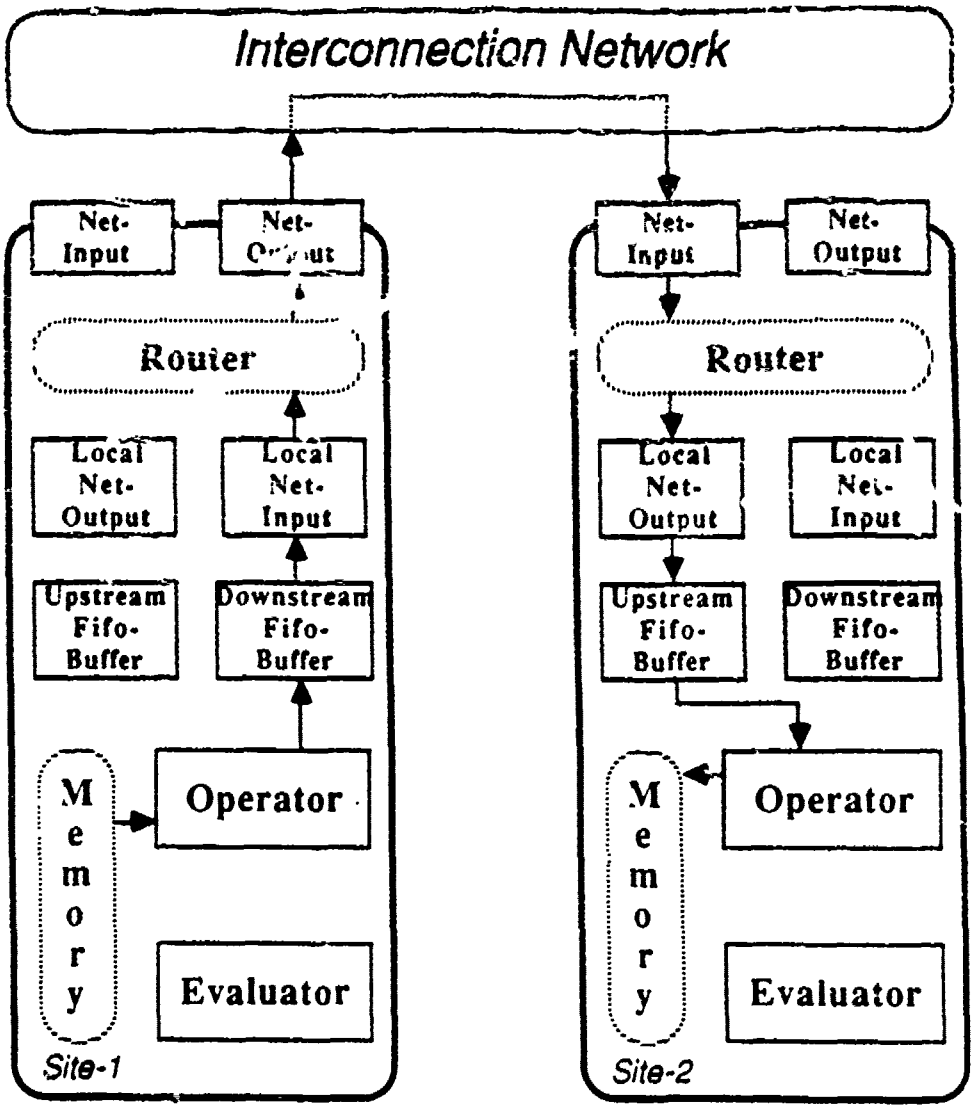


Figure 3: Network component interconnections. Packets travel in the direction marked by arrows. Flow control information flows in the opposite direction.

packet. Each non-local net-output is physically connected to a net-input on a (logically) neighboring site. When available, this net-input accepts the packet, and its router sends the data to the local net-output, if the target has been reached, or to another net-output, if not. This continues until the target has been reached, where the local net-output delivers the packet to the operator (through a fifo-buffer). The operator can then perform whatever operation is specified by the packet, such as storing the value in memory or queueing some operation for the evaluator, for example.

If the packet has more than one target, the router may split it—that is, it may send (essentially) the same packet to several net-outputs. This is called a *multicast* transmission. Each transmitted packet contains a distinct *subset* of the targets of the original packet. The copying operation is done during transmission, one word at a time, as opposed to buffering the entire packet and making copies. If one branch of the multicast is blocked, the net-input sends *pad characters* down the other branches until valid data may be sent down all the paths. The pad characters are thrown away when received by a fifo-buffer.

3.3 Flow Control

Flow control information, in the form of status signals, flows in the direction opposite to packet transmission. There are four distinct status signals, as shown in Table B. The status signals are used to indicate to the upstream component whether the packet or packet terminator can safely be transmitted.

A **'free** signal means that the component is not currently involved in a transmission and is ready to receive data. An **'open** signal is used when the component is involved in a transmission and is ready to receive the next word of the packet. If the transmission becomes blocked for some reason, a **'wait** signal is sent upstream to temporarily halt the flow of data. Finally, the **'abort-request** signal indicates that a potential deadlock condition has been detected and the transmission may be aborted. Details about how these signals are generated and interpreted will be presented in Section 4.

Status	*Meaning*
'free	Available to receive packet.
'open	Packet header has been received; available to receive more data.
'wait	Busy or network is blocked; do not send more data.
'abort-request	Potential deadlock detected.[a]

[a] Only a fifo-buffer may originate the **'abort-request** signal.

Table B: Flow-control signals.

Component	Odd Phase	Even Phase
Net-Input	Latch status from downstream and conditionally open data latch to allow data to flow downstream.	Open status latch to allow status information to flow upstream.
Net-Output	Open status latch to allow status information to flow upstream.	Latch status from downstream and conditionally open data latch to allow data to flow downstream.

Table C: Communication cycle phases.

A *communication cycle* consists of two major phases[5] (see Table C). During one phase, a component latches the status signal from downstream. Based on that signal, it may open its data latch to allow data from upstream to flow downstream. Otherwise, it holds the previously latched data. During the other phase, the component opens its status latch to allow status information (perhaps modified by the component) to flow upstream. The cycles of adjacent network components (e.g., net-inputs and net-outputs) are arranged so that one component is latching the status information while the downstream component is determining the status for the next cycle. Thus there cannot be a race between the latching of data and the status signal which controls it.

3.4 Deadlock Avoidance

The existence of packet multicasts introduces the possibility of deadlock. A packet traveling through the network acquires the use of network resources (e.g., net-inputs and net-outputs) and simultaneously excludes the use of those resources by other packets. Without special attention paid to the possibility of deadlocks, it is possible that resources are consumed to perform the multicast, but completion of the multicast is not possible because the resources acquired are insufficient.

If only unicast transmissions were allowed, this kind of deadlock would not occur. Assuming that a packet cannot be infinitely long, a blocked unicast packet will eventually either acquire the network connection that it needs or be (temporarily) stored at the local site (freeing up any upstream resources for

[5] Any necessary signal serialization would occur within a major phase.

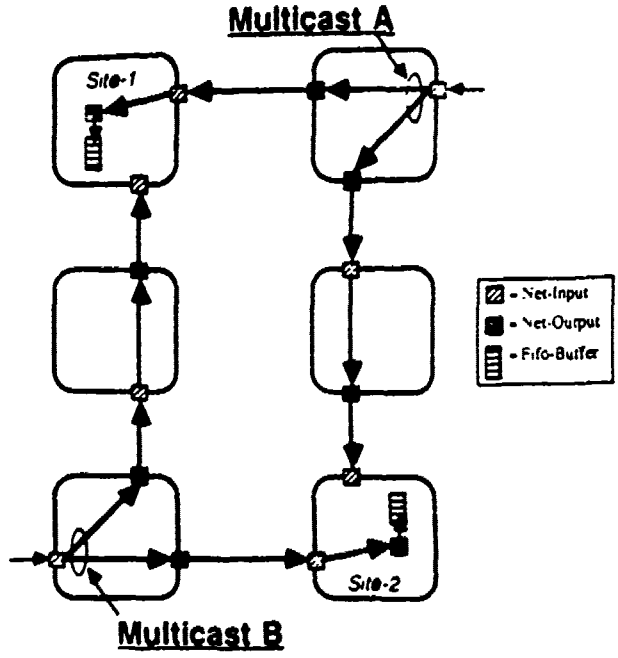


Figure 4: Example of deadlock in a multicast.

this packet). In other words, any resource that is acquired will eventually be released.

Figure 4 illustrates an example of how multicast deadlock can arise. Suppose we have two multicast transmissions, call them $\mathcal{A}$ and $\mathcal{B}$, with common destinations, *site-1* and *site-2*. Suppose that one of the packets from multicast $\mathcal{A}$ has already gained access to the local net-output on *site-1*. A packet from multicast $\mathcal{B}$ has similarly gained access to the local net-output on *site-2*. For multicast $\mathcal{A}$ to continue, it needs to gain access to the local net-output of *site-2*;[6] for $\mathcal{B}$ to complete, it needs to gain access to the local net-output on *site-1*. Also, neither of the multicasts can release the resources it has already required until the transmission is completed. Since each multicast has acquired a resource that the other needs, a deadlock results.

In order to recover from such a situation, the system must:

- Detect a potential deadlock condition, such as the situation described above;
- Back out of the unsafe condition (by aborting one or more transmissions, thereby releasing some set of resources); and

[6]The transmission cannot continue because the net-input cannot send any words until all branches of the multicast are ready to receive it. Since the branch waiting for the local net-output of *site-2* is blocked, none of the branches may proceed.

- Retransmit the aborted packets later, when the network is (hopefully) less congested.

Whenever a packet is split for multicast, the protocol requires that a copy of the original packet (with a complete target list) be sent to the local net-output. This packet will then be stored in a fifo-buffer, so that it may be retransmitted in the case that the current multicast must be aborted due to deadlock.

The packet terminator has two roles in deadlock avoidance. First, a fifo-buffer can detect a potential deadlock if the packet terminator has not been received in a "reasonable" amount of time.[7] Second, the packet terminator indicates to all operators which received the packet what should be done with it. For example, a multicast is aborted by sending the **:abort-packet** terminator downstream—all operators which receive a packet with this terminator will ignore the packet. Also, the operator which receives the copy of the original packet can tell whether it needs to be retransmitted by looking at its terminator. More details will be presented in the next section.

These actions are sufficient to prevent persistent deadlock during multicasts. However, since there is finite storage in the system, a scenario can be constructed in which all the storage becomes committed and no packets can be delivered. The protocol does not prevent this type of resource exhaustion. The assumption is made that the designed capacity of the system is sufficient for its applications.

4 The Protocol

This section provides a detailed description of the behavior of each of the network components. First, however, we present the details of the deadlock avoidance mechanisms, so that the behavior of individual components can be understood in the context of an overall transmission.

4.1 Deadlock Avoidance Mechanisms

The protocol mechanisms which allow deadlocks to be detected and avoided are as follows:

1. If a packet has multiple targets, before a router can split the packet for multicast, the local net-output must be available. This is to insure that a connection to the fifo-buffer is possible, so that the packet may be stored for possible retransmission.

 (a) The local net-output is sent a copy of the packet which contains a complete target list (rather than a subset). This assures that the packet may be retransmitted to all of its targets if the multicast is aborted.

[7] See Subsection 4.1.

(b) If the local net-output is unavailable, then the packet may be sent, but only to a single target. The intent is that a packet sent in this fashion will either visit each target site individually, or will eventually reach a site with an available local net-output and be multicast to the remaining sites on the packet target list.

2. Upon receiving the front end of a packet, the fifo-buffer starts a timeout procedure.[8] If the timeout occurs before the packet terminator is received, the fifo-buffer asserts the **'abort-request** signal upstream on the flow control line.

 (a) When a net-input currently engaged in a multicast receives an **'abort-request** (from a downstream fifo-buffer) *before* it sends the packet terminator, the net-input goes into *abort mode*.

 (b) Net-inputs which are not involved in a multicast ignore the **'abort-request** signal; net-outputs merely pass an **'abort-request** upstream.

3. In *abort mode*, the net-input performs several actions:

 (a) All connected non-local net-outputs are sent the **:abort-packet** terminator, and they are disconnected from the net-input. This signals any downstream operator to ignore the packet when it is received. At this point, only the connection to the local net-output is active.

 (b) The **'open** flow control signal is sent upstream to unblock the packet transmission.

 (c) When the packet terminator arrives at the net-input, the packet terminator that is received is passed on to the local net-output. The **:abort-packet** terminator causes the local operator to discard the packet. The **:end-of-packet** terminator will result in retransmission, if the original target list contained remote (not local) sites.

4.2 Generic Component Description

Next we describe the behavior of individual components. Most of the components are described as finite state machines which have input ports, output ports, and internal state variables. The input and output ports are used to pass packets and flow control information—packets flow downstream, flow control signals flow upstream. The ports and their functions are shown in Table D. Figure 5 shows a "generic" network component, with its input and output ports.

[8] The intent is to determine when the packet terminator has not arrived in a "reasonable" amount of time. This might actually be a timer, where the interval is some function of the expected packet length, or it might be some threshold limit for the number of consecutive pad characters a fifo-buffer will accept. The details are not specified by the protocol documented here.

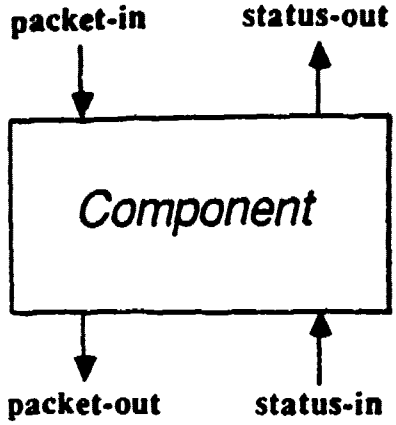


Figure 5: Generic network component.

Port	*Function*
packet-in	Packet data from upstream component.
packet-out	Packet data to downstream component.
status-in	Flow control from downstream component.
status-out	Flow control to upstream component.

Table D: Input and output ports.

The behavior of most of the components can be described in terms of states and transitions between those states (i.e., a state machine). It is often useful to illustrate the states and transitions in a state transition diagram, as in Figure 6. The transitions are labelled with the condition used to trigger the transition, and the status signal to be sent upstream (through the **status-out** port) when the transition is made.

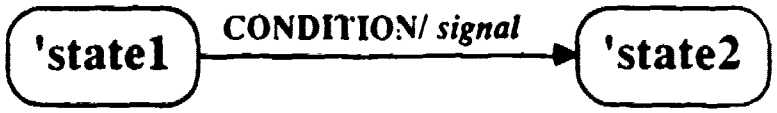


Figure 6: A state transition diagram.

4.3 Operator

The operator sends and receives packets through the network and through the memory it shares with the evaluator. Thus, it has more than one set of ports for

packet communication. To avoid confusion, the ports it uses to communicate with the network are prefixed **network-** (e.g., **network-packet-in**), while the ports used for communication with the evaluator are prefixed **evaluator-** (e.g., **evaluator-packet-in**). Only network communication will be discussed in this paper.

With respect to the network, both the upstream and downstream components of an operator are fifo-buffers. The upstream fifo-buffer queues packets from the local net-output and sends them to the operator. The downstream fifo-buffer queues packets from the operator and sends them to the local net-input.

Two state variables are used by the operator for network communications:

1. **network-buffer:** Used to temporarily store an incoming packet from the network.
2. **network-buffer-status:** Indicates whether the packet in the **network-buffer** has been serviced ('**new** or '**old**).

4.3.1 Sending a Packet

The operator has a queue of *operations*, or requests, which it services in order of arrival. If the head of this queue is a packet to be sent out into the network, and **network-status-in** is '**free**, indicating that the downstream fifo-buffer is ready to accept a packet, the operator sends the packet (with an :**end-of-packet** terminator) through the **network-packet-out** port.

4.3.2 Receiving a Packet

A packet arrival at the operator is signalled by the appearance of data on the **network-packet-in** port. The **network-status-out** port is set to '**open**, which signals to the upstream fifo-buffer to keep sending packet data until the packet terminator arrives. The packet data is stored in the **network-buffer**.

The arrival of an :**end-of-packet** signifies that the packet transmission was successful. **Network-buffer-status** is set to '**new**, signifying that the data in the temporary buffer should be looked at. At some later time, the operator services the packet and sends a '**free** signal to the incoming fifo-buffer (through **network-status-out**), indicating that another packet may be received, and **network-buffer-status** is set to '**old**, so that the packet is not serviced twice.

If the operator notices that some or all of the target addresses of the received packet do not correspond to its own address, the packet is sent back out into the network. This might happen for one of the following reasons:

1. During a unicast transmission, a net-input could not make a connection to the desired net-output. The packet is forced into the local fifo-buffer, so that the operator may resume the transmission at a later time, freeing up the net-input and its upstream components.

2. A multicast transmission was aborted. The local fifo-buffer received a copy of the packet with a complete target list, so that the packet could be retransmitted in case of an abort.

A :local-end-of-packet terminator instructs the operator to accept the packet, as in the case of :end-of-packet, but to ignore any non-local target addresses. This indicates that a multicast was successful, and so does not have to be retried.

The arrival of an :abort-packet terminator instructs the operator to discard the packet. The operator then asserts 'free on network-status-out, indicating that another packet may be received, without setting network-buffer-status to 'new—that is, the packet data in the temporary buffer is never serviced.

4.4 Fifo-buffer

Each site has two fifo-buffers, which have identical behavior but perform slightly different functions. One fifo-buffer is upstream with respect to the operator, and the other is downstream.

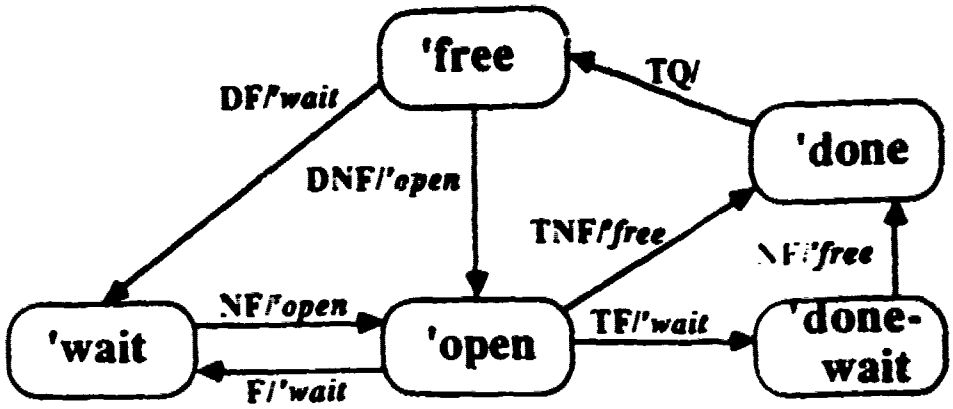
On its output side, the upstream fifo-buffer is connected to the operator, while the downstream fifo-buffer is connected to the local net-input. If the queue is not empty, the fifo-buffer responds to a 'free or 'open signal on the status-in port by removing the oldest item from the queue and sending it through the packet-out port. If a 'wait signal is received, the transmission is temporarily halted until a non-'wait signal appears.

On its input side, the upstream fifo-buffer is connected to the local net-output, and the downstream fifo-buffer is connected to the operator. The fifo-buffer needs to keep track of (1) whether the packet data and terminator have been received and (2) whether they have been placed in the queue. The state diagram of the input side is shown in Figure 7, and the states are described in Table E.

State	*Meaning*
'open	Ready for more data; terminator not received.
'wait	Queue full; terminator not received.
'done	Terminator received, but not yet queued.
'done-wait	Terminator received, but queue full.
'free	Terminator queued, ready for next packet.

Table E. Input states for fifo-buffer

The fifo-buffer begins in the 'free state. Whenever data arrives on the packet-in port, if the queue is not full, the 'open state is entered and 'open is asserted on status-out. If the queue is full, the 'wait state is entered and 'wait is asserted; when space becomes available in the queue, the 'open state



Condition	Meaning
DF	Data arrives, and queue full.
DNF	Data arrives, and queue not full.
F	Queue full.
NF	Queue not full.
TF	Terminator arrives, and queue full.
TNF	Terminator arrives, and queue not full.
TQ	Terminator queued.

Figure 7: Fifo-buffer state diagram.

is entered and 'open is asserted. If the queue becomes full at any point in the transmission, the 'wait state is entered and the 'wait signal is asserted on status-out, so that no more data will be sent from upstream. When space becomes available, the 'open state is re-entered, and 'open is sent upstream to resume the flow of data.

When a packet terminator arrives, if the queue is not full, the 'done state is entered and 'free is asserted on status-out. If the queue is full, the 'done-wait state is entered first, which asserts 'wait until space is available in the queue. Then the 'done state may be entered. When the terminator is actually in the queue, the 'free state is entered, and the fifo-buffer is ready to receive another packet.

Not shown in the state diagram is the timeout procedure mentioned in Subsection 4.1. This is because the details of the timeout procedure are dependent on the implementation. The intent of the timeout is to indicate when the fifo-buffer has been waiting an unusually long time for the packet terminator. When

a timeout occurs, the **'abort-request** signal is sent upstream through **status-out**. The fifo-buffer behavior then continues as described above.

4.5 Net-Input

The downstream component from a net-input is a router, but the values on the **status-in** port are actually originated from a downstream net-output and are passed through the router. If the net-input is local (connected to an operator), its upstream component is a fifo-buffer; otherwise, its upstream component is a net-output (on a logically neighboring site). The states of the net-input are shown in Table F, and the transitions are illustrated in Figure 8. A state variable, **connection**, is used to save the type of the current downstream connection.

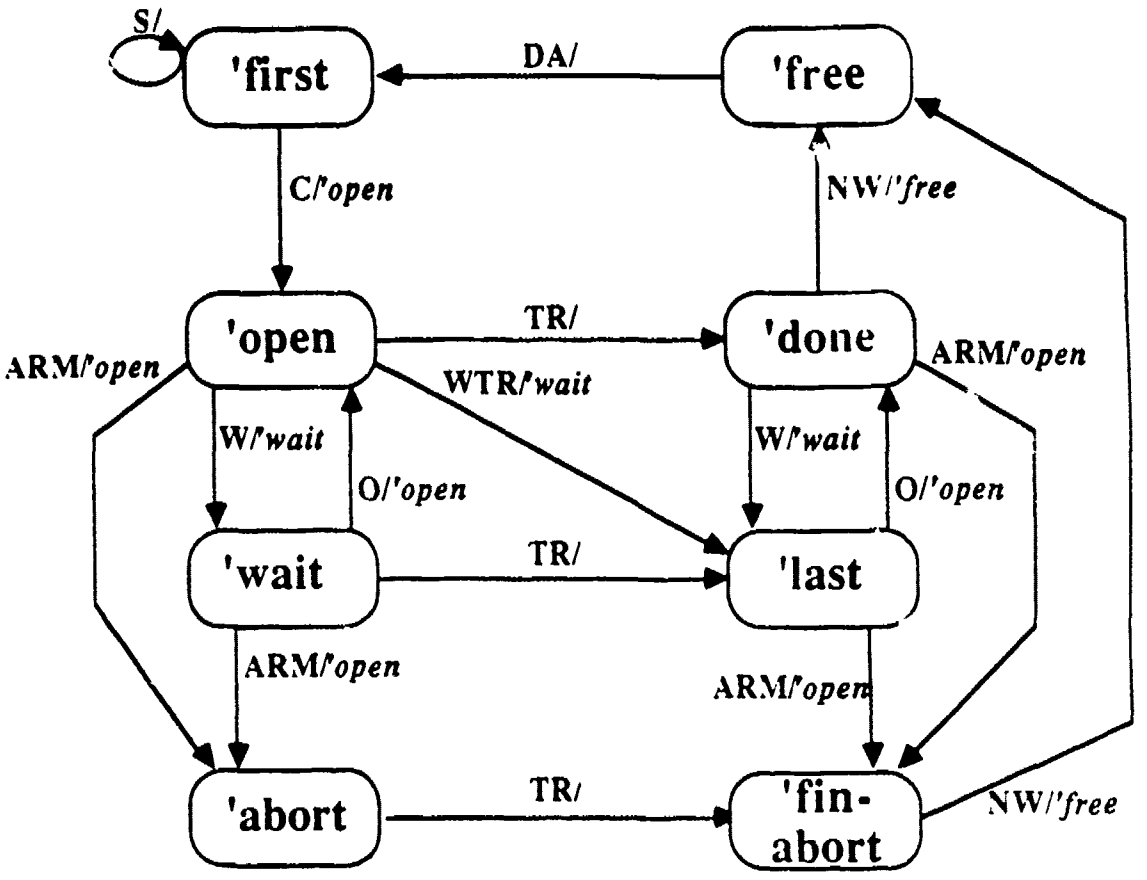
Value	*Meaning*
'first	Packet received, but net-input not yet connected to the network.
'open	Connected to network and packet transmission in progress.
'wait	Downstream requested wait after transmission started.
'done	Terminator received, but not sent.
'last	Downstream requested wait after terminator received, but before it was sent.
'abort	Abort requested from downstream.
'fin-abort	Abort requested, and terminator received.
'free	Idle—remains in this state until the network connection goes free *and* a new packet is received.

Table F: States for net-input.

The net-input begins in the **'free** state, with all its downstream connections free. When the front end of a packet arrives on **packet-in**, it is sent directly to the router, which attempts to make the proper connection based on the packet's target list. If the router is successful, it makes the appropriate connections, begins transmission of the packet to the connected net-output(s), and returns one of the following values on **connection**, which indicates the type of connection that was made:

'unicast All targets of the packet reside on a single site.

'passthru The packet has multiple sites in its target list, but has only been sent to a single net-output. This type of connection indicates that the local fifo-buffer was not available to accept a copy of the packet.



Condition	Meaning
DA	Data arrives.
S	'Seek returned (try again).
C	Connection obtained.
W	'Wait rec'd on status-in.
O	'Open rec'd on status-in.
ARM	'Abort-request rec'd & this is a multicast.
TR	Terminator received.
WTR	Terminator and 'wait received.
NW	Non-wait signal rec'd on status-in.

Figure 8: Net-input state diagram.

'all-remote The packet has multiple sites in its target list, and the router has made connections to multiple net-outputs. The packet's target list contained only non-local sites.

'some-local The packet has multiple sites in its target list, and the router has made connections to multiple net-outputs. The packet's target list included the local site.

If the connection attempt is unsuccessful, because of busy channels, for example, the router returns **'seek**, which prompts the net-input to try again. If the number of unsuccessful attempts exceeds a threshold, the router sends the packet to the local net-output—the local operator will retransmit the packet if any destination in the target list is not local.

A successful connection causes the net-input to enter the **'open** state and to assert **'open** on **status-out**. At this point, several possible transitions can occur. We will first consider the *commit* case, where no **'abort-request** is received and the net-input successfully delivers the packet. Later, we will consider the *abort* case.

4.5.1 Commit Mode

Ignoring **'abort-request** for the moment, two possible events can occur: (1) the packet terminator arrives on the **packet-in** port, or (2) one or more downstream net-outputs send **'wait** over the **status-in** port. The **'wait** state is entered if a **'wait** signal is received; the **'done** state is entered if the packet terminator is received; the **'last** state is entered if both are received. Figure 8 shows the possible transitions among these states. Whenever a **'wait** is received from downstream, **'wait** is asserted on **status-out** to halt the information flow from upstream, as well. The wait condition is cleared when an **'open** signal appears on **status-in**. This indicates that all the downstream net-outputs are ready to receive the packet terminator and causes a transition from **'wait** to **'open**, or from **'last** to **'done**.

If the net-input is in the **'done** state and **'open** is received from downstream, the appropriate packet terminators are sent according to the type of connection:

'unicast *or* **'passthru:** An **:end-of-packet** is sent to the single downstream net-output (local or remote).

'all-remote: An **:end-of-packet** is sent to all the non-local connected net-outputs; **:abort-packet** is sent to the local net-output, because the operator should discard the packet rather than attempt to re-send it.

'some-local: An **:end-of-packet** is sent to all non-local connected net-outputs; **:local-end-of-packet** is sent to the local net-output, so that the operator will ignore the remote addresses in the packet's target list.

After the packet terminator has been sent out, all connections to net-outputs are released, the **'free** state is entered, and the net-input is available to receive the next packet.

4.5.2 Abort Mode

Abort mode is entered if an **'abort-request** is received from downstream before the packet terminator is sent downstream, and the current transmission is a multicast (**'all-remote** or **'some-local**). (**'Abort-request** is ignored on a non-multicast transmission. From this point, we will assume a multicast transmission.)

If the **'abort-request** is received before the packet terminator (i.e., while in **'open** or **'wait**), the **'abort** state is entered. When the packet terminator arrives, the net-input enters the **'fin-abort** state. Alternatively, the **'abort-packet** could arrive after the packet terminator, in which case **'fin-abort** is entered directly from **'done** or **'last**.

Whenever abort mode is entered, the net-input sends an **:abort-packet** to all non-local connected net-outputs and disconnects them. They will, in turn, pass the terminator downstream when possible. The only connection retained is to the local net-output. When the local net-output is ready to receive the packet terminator (i.e., **'open** is received on **status-in**), the net-input passes on whichever type of terminator it received. The two cases are as follows:

:end-of-packet No upstream packets have been aborted, so it is the responsibility of this site to abort the downstream transmissions and to re-transmit the packet. Upon receiving the **:end-of-packet**, the operator will notice some non-local addresses in the packet's target list and will send it back out into the network.

:abort-packet Some upstream site is aborting the multicast and will eventually resend the packet. The operator on this site, then, is instructed to ignore this packet.

The net-input then enters the **'free** state and releases the local connection, ready to receive the next packet.

4.6 Router

The router is responsible for the following:

- Determining to which net-outputs a packet should be sent, based on its list of target addresses, the system routing strategy, and the current availability of net-outputs; and
- Creating, maintaining, and deleting the connections between a net-input and a set of net-outputs, including transmitting data and flow control signals between them.

The router, unlike the other components, is not modelled as a finite state machine—it is conceived as a priority network (implemented in combinational logic, for example). Information about routing and active connections can be thought of as residing in the tables shown in Table G.

Table	*Contents*
preference-table	For each logical output direction, a sorted list of net-outputs to be considered.
input-connection-table	For each net-input, a list of connected net-outputs.
output-connection-table	For each net-output, its connected net-input.
output-status-table	For each net-output, its transmission status.

Table G: Routing tables.

The first words of the packet are always the target list. As each target is received, the router makes an appropriate connection to a net-output and sends that address downstream. The routing (for each target address) takes place in a single communication cycle,[9] so there is no additional delay introduced by the router.

If there is only one target, the router makes the connection (see below) and returns **'unicast**. If there is more than one target, the router checks the status of the local net-output. If the status is **'free**, then the appropriate connections are made and either **'all-remote** or **'some-local** is returned. If the local net-output is not **'free**, then a single connection is made based on the first target on the list (ignoring the other targets), and the returned connection value is **'passthru**.

Making a connection involves determining the logical "direction" (e.g., up or down) of the target from the local site, then determining which net-output should be used for that direction, and finally updating the connection tables and starting the packet transmission.

Determining the logical direction depends on the network topology and is usually straightforward. For example, a grid or torus requires only some arithmetic comparisons between the target address and the local address to get Up, Down, Right, Left, or some combination of these. A hypercube, on the other hand, requires an exclusive-OR operation to see which bits in the destination address are different than the local address. Equally simple operations can be envisioned for most other network topologies, as well.

[9] See Subsection 3.3.

Once the logical direction is determined, the router looks in the **preference-table** for a list of net-outputs to consider. This table implements the system routing strategy and is determined when the system is built. It lists, in decreasing order of preference, all the net-outputs that might be used to send a packet in a given logical direction. The router checks all the status of each of these, in turn, until an available net-output is found. If none is found, then the connection fails, and **'seek** is returned to the net-input.[10] Examples of routing strategies which may be implemented by the routing table are (1) try all net-outputs, starting with the closest to the target, (2) try only one net-output (static routing), and so forth.

During the transmission, the router is responsible for passing flow control information from the net-outputs to their connected net-inputs. If a net-output, for example, asserts **'wait** on its status line, the router must relay that signal to the net-input which is connected to it. Also, the router cannot pass the net-input an **'open** signal until *all* of its downstream net-outputs are in a non-wait state. The **input-connection-table**, **output-connection-table**, and **output-status-table** are useful for these types of operations.

4.7 Net-Output

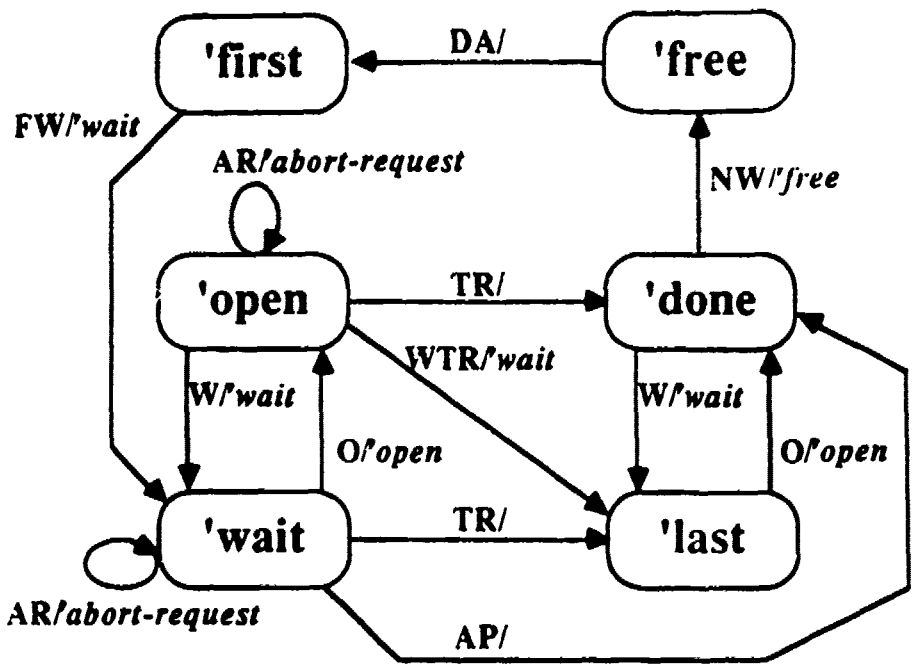
The upstream component of a net-output is always a net-input. On the downstream side, the local net-output is connected to the fifo-buffer which delivers packets to the operator, while a non-local net-output is connected to a net-input on a logically neighboring site. The net-output states are listed in Table H, and the transitions are illustrated in Figure 9.

State	*Meaning*
'first	Packet received, but not yet sent.
'open	Packet transmission in progress.
'wait	Downstream requested wait.
'done	Terminator received, but not sent.
'last	Downstream requested wait after terminator received, but before it was sent.
'free	Terminator sent, ready to receive next packet.

Table H: States for net-output.

The net-output is initially in the **'free** state. When a packet arrives on **packet-in**, it enters the **'first** state. If its downstream component (either a

[10] Note that, in the case of a multicast, partial finds (in which only a subset of the targets can be assigned to net-outputs) must be forced to fail (by sending an :abort-packet terminator over the connections made thus far), or the operator would not know which parts of a multicast to retransmit in case of an abort.



Condition	Meaning
DA	Data arrives.
FW	'Free or 'wait rec'd on status-in.
W	'Wait rec'd on status-in.
O	'Open rec'd on status-in.
AR	'Abort-request rec'd on status-in.
TR	Terminator received.
WTR	Terminator and 'wait received.
AP	:Abort-packet terminator received.
NW	Non-wait signal rec'd on status-in.

Figure 9: Net-output state diagram.

net-input or a fifo-buffer) has placed 'wait on the status-in port, the net-output asserts 'wait on status-out, which inhibits the upstream net-input from sending anything else. When the downstream component becomes ready to accept the packet, it will assert 'free.

When a 'free signal is received from downstream, the net-output transmits the packet and enters the 'wait state, asserting 'wait on status-out. The net-output remains in the 'wait state until an 'open signal is received from downstream.

The net-output then enters the 'open state, sending an 'open signal to the upstream net-input (via the router). Things then continue much the same as in the net-input. 'Wait is entered if the downstream component requests a wait and the packet terminator has not arrived. 'Done is entered when the packet terminator arrives; 'last is entered if a wait is requested from downstream after the terminator arrives. If an 'abort-request is received from downstream before the packet terminator arrives, it is relayed to the upstream net-input. If the packet terminator has already arrived, then the 'abort-request was premature and is ignored.

Then the net-output sends the packet terminator, when the downstream component is ready to accept it, and enters the 'free state. When the downstream net-input accepts the packet terminator and responds by asserting 'free, the net-output asserts 'free on its status line. The upstream net-input will then release the connection, and the net-output becomes available to receive the next packet.

5 CARE Implementation

In this section, we provide an overview of the implementation of the protocol in the CARE simulation system. CARE is a library of functional modules and intrumentation built on top of an event-driven simulator [7], which is used to investigate parallel architectures. The typical CARE architecture is a set of processor-memory pairs (*sites*) connected by some communications network, though it can also be configured to represent a system of processors communicating through shared memory. The behavior and relative performance of CARE modules can easily be changed, and the instrumentation is flexible and useful in evaluating the performance of an architecture or in observing the execution of a parallel program.

CARE is implemented using Flavors—an object-oriented extension of Zetalisp [8]. Roughly speaking, each component described in Section 2 is implemented as an object (an *instance* of a flavor). (One notable exception is the router—its functions and tables are assumed by the *site* object, rather than implemented as a separate component. Also, the memory at a site is not explicitly represented as an object, but exists implicitly in the simulator.) Associated with each object is a set of *instance variables*, used to hold state information, and

a set of *methods*, procedures used by the object to respond to messages from other objects.[11] The instance variables loosely correspond to the ports and state variables used to describe the protocol in Section 4. In particular, each of the components which are described in terms of a state machine has a instance variable, **packet-status**, which hold the current state of the component.

These objects communicate through shared structures called *vias*, which represent unidirectional data paths. These are the "wires" which connect the components' "ports." Asserting a value on the sending end of the via immediately (in simulated time) triggers an event for the object at the other end. Therefore, a via can be considered a zero-delay wire which can transmit any arbitrary value (not just single bits).

The simulation is functional,[12] rather than circuit-level, and event-driven, rather than clock-driven, because cycle-by-cycle simulation of a parallel machine would be extremely time-consuming, especially when the number of processors is large. For this same reason, we do not wish to model the transmission of a packet one word at a time. Instead, a packet is represented by two distinct parts, one representing the contents of the packet, and the other representing the packet terminator. In the following discussion, *packet* will refer to the first part (representing the front edge of a "real" packet), and *packet terminator* will refer to the terminator part.

In the simulation environment, explicit packet terminators allow us to (1) implement the deadlock avoidance mechanisms described earlier, and (2) model the transmission of a packet through the network in terms of its front edge and its back edge. In this way, if the time between the transmission of the packet (front edge) and its terminator in the simulator is the same as the transmission time of the packet in a real machine, we can accurately model the transmission of the packet without explicitly representing every word.

In the following subsections, we describe how the protocol is implemented in terms of objects, packets, and packet terminators.

5.1 Operator

The time required to transfer a packet from the operator to a fifo-buffer (one word at a time) would be proportional to the size of the packet. To model this,

[11] Objects and messages are only a software tool used by the simulator. Sending *messages* between objects in the simulator has no particular correspondence to sending *packets* between components in the target machine.

[12] The simulation is functional, in the sense that not every aspect of the hardware is simulated in detail. Some aspects are simulated by register transfer level behavior, while other aspects have only a functional description. For example, the execution of application code by the evaluator is not simulated at all—it is directly executed by the host machine. However, timing information for the execution of application code, based on measurements and estimates, is used to assure that the simulation is reasonably faithful to the execution of a "real" machine.

the operator delays an appropriate time between sending a packet and sending its terminator.

Because storage in the simulated fifo-buffer is in terms of packets, rather than bytes[13], there will be no wait signals received from the downstream fifo-buffer. Therefore, merely delaying for a time proportional to packet size is sufficient.

A CARE operator receives a packet as described in the protocol Note that the time between receiving the packet and its terminator is dependent on the size of the packet plus any delays encountered on its transmission path.

5.2 Fifo-buffer

In the simulator, the amount of storage in the fifo-buffer may be set at run time.[14] Each packet or packet terminator takes up one space in the buffer, no matter what its actual size. In particular, the buffer cannot fill up in the middle of accepting a packet, so the 'wait state will never be entered. Thus the operator, which feeds data into a fifo-buffer, does not have to deal with any waiting time in the middle of transmitting a packet, as described above. This simplifies the implementation of the protocol, at the expense of a slight loss of fidelity in the simulation.

On the output side, however, the simulated fifo-buffer is more complex than the protocol indicates. If a packet is being output from the queue, the fifo-buffer must introduce a delay between the packet and its terminator to model the packet transit time. However, the transit time is not merely proportional to packet size, because downstream blocking could cause arbitrary delays in the transmission.

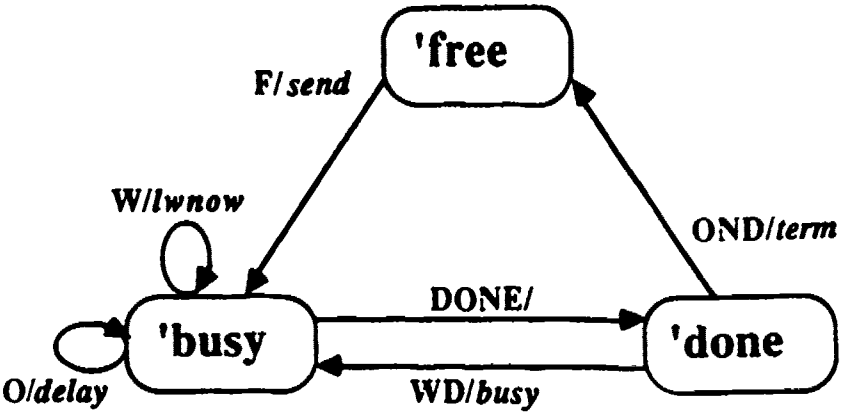
The simulated fifo-buffer output transitions are shown in Figure 10. In this case, the transitions are labelled with conditions and *actions*, rather than flow control signals, Some additional instance variables for the fifo-buffer are required to implement the output function. They are:

1. **transmission-status:** State of packet output.
2. **delay:** Accumulated time spent waiting.
3. **last-wait:** Event time when last 'wait was received.

Initially, **transmission-status** is 'free. If the downstream component requests data (**status-in** goes to 'free) and the queue is not empty, the top of the queue, which must be a packet, is placed on the **packet-out** via. **delay** is set to zero, and **transmission-status** goes to 'busy. Also, **transmission-status** is scheduled to go to 'done at a time that is proportional to packet size.

[13] See subsection 5.2.

[14] By setting the care:***buffer-size*** variable to any positive integer, or to nil, which means "unbounded."



Condition	Meaning
F	'Free rec'd on status-in.
W	'Wait rec'd on status-in.
O	'Open rec'd on status-in.
DONE	'Done event.
WD	'Wait rec'd and [delay nonzero OR last-wait non-nil].
OND	'Open rec'd and [delay = 0 AND last-wait = nil].

Action	Meaning
send	Send packet, schedule 'done for now + transmission-time.
lwnow	Last-wait = now.
delay	Delay = delay + (now - last-wait); Last-wait = nil.
busy	Schedule 'done for now + delay; Last-wait = nil.
term	Send terminator.

Figure 10 Implemented fifo-buffer output state diagram.

If no 'wait signals are received from downstream while the transmission is 'busy, then the transmission will be done after the packet transit time has elapsed, and the packet terminator will be sent as soon as the downstream component is ready to receive it.

However, if 'wait is received during 'busy, last-wait is set to the current time and waiting is set to t. If 'open is received during 'busy, the time spent waiting is added to delay and waiting is set to nil.

If 'open is received when transmission-status is 'done, and delay is non-zero, then 'busy is entered again, 'done is scheduled for the current time plus the accumulated delay, waiting is set to nil, and delay is set to zero. Alternatively, if waiting is t and delay is zero, then 'done has occurred in the middle of a wait; 'busy is entered, waiting is set to nil, and 'done is scheduled for the current time plus the difference between now and last-wait.

Finally, when 'transmission-status is 'done, delay is zero, and waiting is nil, the top item of the queue (which must be a packet terminator) will be sent. Then transmission-status becomes 'free, and the fifo-buffer is ready to respond to the next data request.

All of this is to ensure that the time between the packet and its terminator is dependent on the packet size plus any network delays along its path. The other components, net-inputs and net-outputs, do not require this added complexity on the output side. They will either maintain the current time separation or add to it due to downstream blockages, so there is no chance of their sending the packet terminator prematurely.

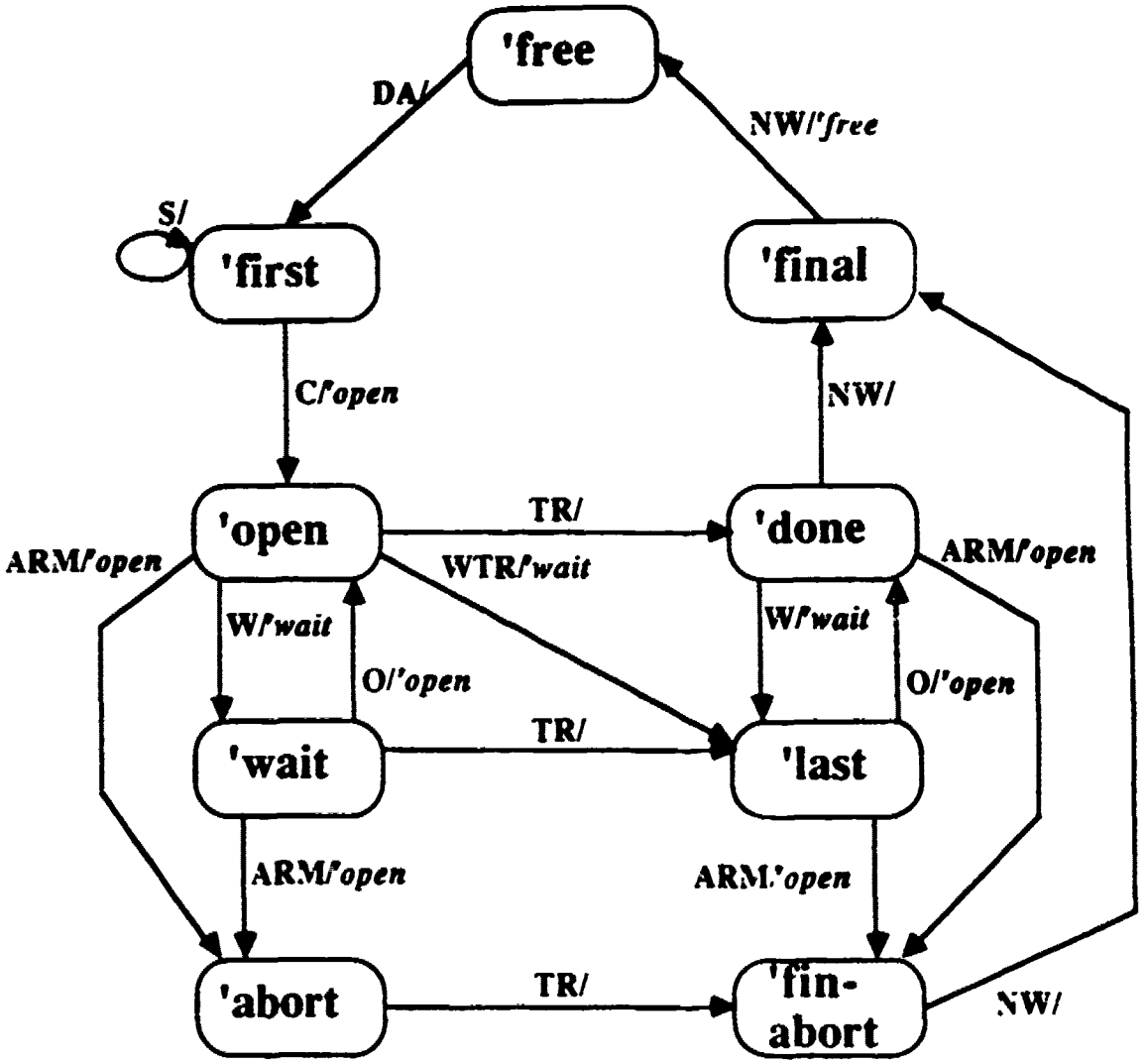
5.3 Net-Input

The main differences between the implementation and protocol concerning the net-input stem from the fact that there is no explicit router in CARE. Each net-input, then, communicates with the *site* which owns it (see Section 2), rather than with a downstream router. The communication is done by passing Flavors messages, rather than asserting data on vias—thus, there is no packet-out instance variable, and status-in is not a via.[15]

To connect to net-outputs, the net-input sends a :connect message to the site. The site then performs the routing and makes the connections as described in Subsection 4.6, returning either 'seek or the type of connection made. Also, the site relays flow control information from the connected net-outputs by setting status-in.

Other site methods used by the net-input include :disconnect-remote, which releases the connections to all net-outputs except the local one, and :send-all, which transmits a packet or terminator to all connected net-outputs. (:Send-local and :send-remote transmit to a subset of connected

[15] Vias must connect two distinct objects; status-in may be connected to any group of net-outputs at a given time, so using a via is not appropriate.



Condition	Meaning
DA	Data arrives.
S	'Seek returned (try again).
C	Connection obtained.
W	'Wait rec'd on status-in.
O	'Open rec'd on status-in.
ARM	'Abort-request rec'd & this is a multicast.
TR	Terminator received.
WTR	Terminator and 'wait received.
NW	Non-wait signal rec'd on status-in.

Figure 11. Implemented net-input state diagram.

net-outputs.)

There is a potential software race in the simulator, which is avoided by adding an additional state in the net-input state machine description. If the net-input is in the 'done state and notices that none of the downstream net-outputs has asserted 'wait, it sends the packet terminator. However, there might be a simulation event scheduled for the same time slot in which one of the net-outputs receives a 'wait and propagates it upstream. In a real machine, this means that the terminator would not have been sent, but there is no way to "undo" the first action by the simulator.

Thus, instead of sending the terminator from the 'done state, the net-input schedules a transition to the 'final state two event-times later. This allows time for all the possible 'wait signals to be handled during the same event. When the 'final state is entered, the state of the connected net-outputs is again examined. If none of them are blocked, the packet terminators are sent immediately (in simulation time), and the 'free state is entered. Any 'wait signal which could arrive at that same instant would be too late to block the transmission in a real machine. The implemented version of the net-input state machine is illustrated in Figure 11.

5.4 Router

As mentioned earlier, there is no explicit router object in the CARE implementation. There are, however, site functions and methods which perform routing in response to a :connect message sent by a net-input.

The :find-direction method determines the logical direction of a target, given its address. This is defined as a method, rather than a function, because this operation is topology-dependent. In Flavors, we can define a specialized *site* object for a particular topology by changing this one method and inheriting the remaining behavior from the generic site definition.

The setup-targets function examines the target list, makes the connections, and copies the packet, as needed. Finally, the make-connections function is resposible for actually setting up connections and sending the packet downstream.

5.5 Net-Output

In the CARE implementation of the net-output, there is no explicit status-out instance variable for sending flow control information upstream. Instead, messages are sent to the *site*, as above, which updates the status table for the particular net-output and relays the information to the connected net-input. There are :wait, :open, :abort-request and :free methods defined for the site for this purpose. Also, because packet input can come from any of the net-inputs on the site, packet-in is not implemented as a via

Finally, on the initial transition into the **'wait** state (from **'first**) the net-output sends a **:first-wait** message, which updates the status table but does not trigger an event for the upstream net-input. This prevents unnecessary simulator events used to propagate the **'wait** signal upstream; they are unnecessary because the net-input will not send anything else until the net-output sends an **'open** signal.

5.6 Results

Variants of this protocol have been used for many CARE simulations over the course of several months. Though the performance has not been extensively measured, the protocol appears to offer reasonable performance over a range of network loads. Deadlocks and lost packets do not occur, even when the network is extremely congested. Thus, our experience with the protocol indicates that it offers efficient and robust one-to-one and one-to-many interprocessor communication.

6 Conclusion

A protocol for high-performance interprocessor communication has been presented. This protocol supports dynamic, cut-through routing with local flow control, which allows high-throughput, low-latency transmission of packets. In addition, multicast transmissions are allowed, in which a packet is sent to several targets using common resources as much as possible.

The protocol also prescribes mechanisms for detecting and avoiding deadlock conditions due to resource conflicts during multicast. In particular, a copy of the packet is saved before it is split, special packet terminators are used to abort transmissions and trigger retransmissions, and random timeout intervals are used to detect potential deadlock conditions.

Finally, the implementation of this protocol in the CARE simulation system is described. Explicitly representing a packet as the front edge and the terminator allows accurate modelling of word-by-word packet transmission in a functional, event-driven simulator. Also, the success of the implementation indicates that this is a reasonable protocol for interprocessor communication.

References

[1] Tse-yun Feng. A survey of interconnection networks. *Computer*, 12–27, December 1981.

[2] V. Ahuja. *Design and Analysis of Computer Communication Networks.* McGraw-Hill, 1982.

[3] P. Kernami and L. Kleinrock. Virtual cut-through: a new computer communication switching technique. *Computer Networks*, 3:267, 1979.

[4] M. Arango, H. Badr, and D. Gelernter. Staged circuit switching. *IEEE Transactions on Computers*, C-34(2):174–180, February 1985.

[5] P. Kermani and L. Kleinrock. A tradeoff study of switching systems in computer communication networks. *IEEE Transactions on Computers*, C-29:1052, December 1980.

[6] Richard W. Watson. Distributed system architecture model. In *Distributed Systems—Architecture and Implementation*, chapter 2, pages 10–43, Springer-Verlag, 1981.

[7] Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. *An Instrumented Architectural Simulation System*. Technical Report KSL-86-36, Knowledge Systems Laboratory, Stanford University, January 1987.

[8] Sonya Keene and David Moon. Flavors. object-oriented programming on Symbolics computers. In *Common Lisp Conference*. 1985.

Knowledge Systems Laboratory
Report No. KSL-87-07

January 1987

Considerations for Multiprocessor Topologies

Gregory T. Byrd†

Department of Electrical Engineering
Stanford University
Stanford, CA 94305

Bruce A. Delagi

Worksystems Engineering Group
Digital Equipment Corporation
Maynard, MA 01754

This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875.

†G. Byrd is supported by an NSF Graduate Fellowship, with additional support provided by the EE Dept.

Considerations for Multiprocessor Topologies*

Greg Byrd†
Knowledge Systems Laboratory
Stanford University
Stanford, CA 94305

Bruce Delagi
Worksystems Engineering Group
Digital Equipment Corporation
Maynard, MA 01754

Abstract

Choosing a multiprocessor interconnection topology may depend on high-level considerations, such as the intended application domain and the expected number of processors. It certainly depends on low-level implementation details, such as packaging and communications protocols. We first use rough measures of cost and performance to characterize several topologies. We then examine how implementation details can affect the realizable performance of a topology.

1 Introduction—Design Constraints and Opportunities

The base for development of general purpose multiprocessor systems as for computer systems today generally is given by the design constraints and opportunities established by evolving semiconductor design and manufacturing processes. The VLSI design medium brings a new perspective on cost: switches are cheap; wires are expensive. In modern microprocessors, communication costs dominate those associated with logic. Power and cooling budgets are spent driving wires and overwhelmingly, chip area is dedicated to wiring rather than logic [17] To an increasing degree, the dominant delays are associated with driving lines rather than the accomplishment of logic functions per se. One implication is that, all other things being equal, smaller, simpler processors can be expected to have shorter operation cycles than larger, more complex designs [18]. They are also likely to be available in a more recent, higher performance base technology.

At the system level, the consequence of relatively expensive communication is that performance is enhanced if the design establishes that whenever a lot of information has to move in a short time, it does not have to move far. Significant locality of high bandwidth links is a goal. Among the highest bandwidth links in a computer system is that connecting the processor and memory. Early computer systems separated these pieces and put a bottleneck between them to accommodate the packaging realities of the time. processors were implemented with electronic means, memory with magnetic, and their power requirements and EMI characteristics were best dealt with separately There are new realities now: close coupling of processors with local memory is preferred

With these design constraints in mind, we consider a multicomputer implementation based on a set of processor/memory pairs connected by a communications topology. Many topologies have been proposed [8] and have been compared in terms of theoretical cost and performance measures [16]. We argue, however, that the realizable performance of these topologies are closely linked to details of system packaging

2 Interprocessor Connection Topologies

Connection schemes between processing sites can be compared with respect to their cost and performance as a function of the number of sites connected. For a particular connection scheme, if the cost grows no faster than the number of sites and the performance grows at least as fast, that scheme can be described as *scalable* A rough measure of cost is the number of input-output ports required for connection. A rough measure of performance is the number of links in the topology divided by the largest number of links that must be traversed, and thus occupied to accomplish a transmission, in order to get from one node in the

*This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875.

†Supported by an NSF Graduate Fellowship and by the Stanford Dept. of Electrical Engineering.

network to another. This indication of the bound on the number of independent, concurrent transmissions we will call the *concurrency* of the network.

For some topologies, the concurrency of a network may *understate* performance as actually experienced in a given application: to the extent that there is locality of reference in transmissions, the number of links actually traversed may be better approximated by a constant than some function of the number of connected sites. Network concurrency may also *overstate* performance of one topology with respect to another: to the extent that the time to traverse links is not the same for all topologies, those that have non-uniform link costs (perhaps due to physical distance considerations applied to the realized lengths of links) will deliver less performance than the concurrency measure suggests. This is because in these cases, logical adjacency due to high dimensionality is merely apparent—embedding the topology in the dimensionality of space available tends to incur just those expenses related to physical distances that the topology was expected to eliminate.

2.1 Topologies With Scalable Concurrency

Several topologies are shown in Table 1 which have scalable concurrency. As the number of sites is increased, the network grows enough to support the consequential additional traffic. In fact, by this measure of performance, the last three of these four topologies scale performance equally well. However, as will be described, there are other considerations to weigh.

In the crossbar and completely connected topologies, the number of ports, a first approximation to cost, grows quadratically with the number of nodes in the network. Weighing cost and concurrency, then, we might prefer the banyan and boolean k-cube (also known as "hypercube") topologies.

By these measures, there does not seem to be a clear-cut choice between the banyan and the hypercube. A more sophisticated measure of cost would take into account the area required for laying out the topology in a plane [11]. The banyan may have a slight edge in this category[1], but both layouts require relatively long wires, which is undesirable if link transit time dominates switching time.[2]

A major difference between the two topologies is that switching and routing are centralized at the processor in the hypercube, whereas the switching in the banyan is distributed throughout the network. To the extent that storage is required at the switch (as in [3]), it becomes more economical to centralize the switch and utilize the local storage of the processor. For this reason, we prefer the hypercube.

2.2 Topologies With Scalable Cost

There are alternative topologies not as richly connected as those just considered. The topologies in Table 2 all have fixed degree connectivity, so they all have scalable cost as measured by port count. Unfortunately, none of them has scalable concurrency. So, at least among the ten representative topologies discussed, there is no topology that has cost-performance characteristics intrinsically superior to all the others.

Concurrency for the ring and the bus topologies does not increase at all as the number of processors increases. Given no guarantee of transmission source to target locality, these seem unsuitable for systems with a large number of processors (e.g., > 100).

The perfect shuffle and cube-connected cycles (CCC) topologies emulate the $O(\log n)$ latency of the hypercube, but the number of links is linear with the number of processors, so concurrency does not scale. Also, if we measure cost in terms of layout area, the cost of the perfect shuffle ($O(\frac{n^2}{\log^{3/2} n})$) and CCC ($O(\frac{n^2}{\log^{3/2} n})$) [15] do not scale and so will not be considered further.

The tree, grid, and torus topologies all have fixed degree connectivity and have the optimum $O(n)$ area requirement.The tree has a slightly better capacity measure and a lower latency bound. Note, however, that the tree provides no alternate communication paths (useful in network balancing and defect tolerance) and has a bottlenecking root.[3] Connections might be added to provide alternate paths, but, as we will see in the next section, physical link considerations may make the grid or torus a better choice.

[1] The area required to lay out a hypercube in a plane is $O(n^2)$ [2], where n is the number of processors. Since "banyan" actually denotes a class of interconnections it is difficult to make a general statement about its layout. However, let us consider a particular banyan network, the omega network [10], which is $\log n$ stages of perfect shuffle connections. The perfect shuffle has area $O(\frac{n^2}{\log^{3/2} n})$ [15], so we would expect $\log n$ perfect shuffles to require area $O(\frac{n^2}{\sqrt{\log n}})$, which is a slightly better bound than for the hypercube. Other types of banyans, with different fan-in, fan-out, and connectivity characteristics might have even smaller bounds.

[2] See Section 3.

[3] We might be able to deal with this by increasing the bandwidth of the links as we proceed toward the root, for example with "fat trees" [12].

3 Link Costs—Examining The Free Lunch

Most studies of topologies assume a constant cost for link traversals as the number of links increases. This is a useful approximation if the time to drive and receive link signals is constant with link length and large compared to signal transit time on the link. However, this is increasingly not a good assumption both as the underlying feature size of the component technology decreases and as we consider larger numbers of sites in a system. Given a fixed circuit feature size, topologies with scalable concurrency, as discussed in Section 2.1 suffer increased link lengths and thus longer signal transit times—with possibly increasing drive times—as the number of processors increases. Alternatively, given a fixed volume of circuits in these topologies and decreasing circuit feature size, the number of processors in the system increases but so does the ratio between link lengths and feature size. Thus relative to the circuit delay times which are dependent on (and decrease with) circuit feature size, the link transit times become increasingly a more important consideration.[4]

Topology has to be viewed as a dependent variable determined principally by the packaging technology of the system. As an example, consider the recursive-H layout for the binary tree (Figure 1) under the assumption that link transit time dominates switching time. Now consider the grid in Figure 2, which can be laid out in the same area. If transit times dominate, then shorter links and more switching sites will likely shorten the point-to-point communications cycle time and improve the realized capacity of the network.[5] Furthermore, additional data paths allow dynamic routing of messages, and additional computing resources make the grid potentially more powerful than the tree.

Though the torus appears to suffer from extremely long wires which "wrap around" the edges, a simple renumbering of the processors in a grid brings each one within two hops of its logical neighbors[6] (see Figure 3). Thus, we can effectively create a torus by changing the routing algorithm of a grid. Alternatively, we could keep the original torus connections and lay out the processors as in Figure 3(b), resulting in links which are at most twice as long as those for a grid. In the remainder of the paper, we will speak of the grid bearing in mind construction of the torus in these terms.

4 A Packaging Example

We are now faced with two topologies: one with scalable performance—the hypercube—and one with scalable cost—the grid. The arguments presented above suggest that, all else being equal, the communication cycle time for the hypercube would be greater than that of the grid, due to its long links. Even so, the average message latency of the hypercube may still be smaller, due to its high connectivity. To get a better understanding of the relative performance of the two systems, we should examine how they might actually be implemented in near-future technology.

In the mid-1990's we would expect a 0.5-μm MOS fabrication process to be available [7]. We will assume that the complexity of our processor is comparable to today's typical 32-bit microprocessor. The MicroVAX 78032 chip [4], for example, is implemented in 3-μm technology; it measures about 8.5 mm on a side. Using 0.5-μm technology, we could expect a similar processor to require around 1.5 mm on a side. Let us allow 256K bytes (2M bits) of local memory for our processor. Fujitsu's megabit RAM using 1.4-μm technology takes 54.7 mm^2 [6]. If the dimensions of the Fujitsu chip are about 10 mm by 5.5 mm, then a 0.5-μm version would be 3.6 mm by 2.0 mm. Two of these (since we want 2M bits) would be around 3.6 mm by 4 mm. As an approximation, then, each processing element, including a processor, 256K bytes of local memory, and switching and routing circuitry could be expected to fit onto a 5 mm x 5 mm piece of silicon.

Even as devices shrink, die sizes continue to grow. By the mid-90's, the state-of-the-art chips may be as large as 15 mm on a side. Each chip would be expected to have 400–600 I/O pads [14]. Therefore,

[4] The dependence of communication delays on signalling lengths as circuit feature size decreases depends on assumptions made on the thickness and thus the resistivity of associated interconnects. Uniform scaling leads to relative signalling times that increase quadratically with distance [19]. Detailed analysis of the equations of voltage and current in VLSI wire implementations (including consideration of the non-linear characteristics of signal drivers) demonstrated linear dependences [1] but were done assuming that the interconnect (and field oxide) thicknesses did not decrease at all while all other dimensions scaled with the circuit feature size of the technology [17]. Another approach imagines a hierarchy of interconnect of increasing thicknesses with distance [13] to achieve signalling times that grow only with the logarithm of the distance. Yet another approach accepts resistive links but given control over both minimum and maximum wire lengths and use of high impedance receivers, notes that it is possible to counter dispersive losses with reflective voltage doubling at the receiving end of a point to point link [9].

[5] The assumption made here is that the message routing is relatively independent of the computing activities at a processing site, so there is no penalty associated with being routed at a processing site rather than a switch.

[6] This approach is attributed to R. Zippel.

we could put up to nine processing sites on a single die.

The dice could be flip-mounted on a silicon [5] or ceramic [9] substrate with thin-film transmission lines and integrated capacitors. In [9], the maximum length for 5-μm-thick lines is around 20 cm, so we will assume a 10x10 cm module size, on which we can easily place up to 36 dice. We will assume on the order of 1000 I/O pins per module [5].

Consider first packaging a (32x32) 1024-element octal grid, in which each processor is connected to eight neighbors. With nine processors (arranged as a 3x3 grid) on a die, 32 (bi-directional) communication links must come off the chip through the I/O pads, so no more than 18 pads could be used per channel. A module can carry 324 processors, arranged as an 18x18 grid. The entire system, then, could fit on four modules (with room to spare). The communications links from two sides of the 18x18 grid (105 bidirectional channels) must go off-module. Thus, each channel could use 10 pins—one pin for clock and status information and four for data, in each direction.

Now consider a 1024-element hypercube (a "10-cube"). To allow for more complex wiring and easier packaging, we will assume that each die contains eight processors, and each module will hold 32 dice, for a total of 256 processors per module. (Extra space might be used to provide redundant processors for fault tolerance.) Again, only four modules are required to package all 1024 processors. Each processor has ten bidirectional links to its logical neighbors. If the eight processors on a die are wired as a 3-cube, then seven channels from each processor must go off-chip. Five of these channels are connected to other processors on the same module, but two must go off the module. With only $\sim$ 1000 I/O pins for 512 bidirectional channels, it appears that a 1-bit combined control/data stream is all that can be supported for the hypercube communications. If we decrease the number of processors per die to four (and possibly add more memory), we can use separate wires for control and data but the wires will be longer.

Note that in both cases the module pin-out is the limiting factor for channel width, rather than the chip pin-out. If more off-module I/O pins are available, things will look better, but there will still be around a 5-to-1 ratio of the number of required off-module channels in the hypercube as compared to the grid. As mentioned before, the average interconnect length for the grid will be much shorter than that for the hypercube. Therefore, the grid offers shorter (i.e. faster) and wider communication paths than the hypercube when implemented in projected near-future technology.

5 Beyond Topology

As the previous example indicates, the electrical and physical characteristics of the circuit packaging in a system may dictate the scheme used to wire the nodes together. In addition, the communications protocol, that is, the actual signalling on the links are an important component of achievable performance. There are many relevant details—for example:

- Dynamic routing, selecting available links as needed, is useful in balancing load and thus allows more of communication resources of the system to be well used throughout a computation.
- Cut-through routing, making a routing decision on the fly as a packet is received, reduces buffer requirements in the system and minimizes latency experienced in network transit.
- Local flow control, signalling transmission delays back to the source based on local blockage information, together with single "word" buffering and transmission validation at each network input and output port allows the source to complete a validated transmission in a time that does not depend on the size of the network.
- Point to point multicast, sending (approximately) the same packet to multiple targets using common resources to the largest degree possible—coupled with dynamic, cut-through routing, flow control, and word level buffering and transmission validation—provides "virtual busses" precisely as and when they are needed.

A point-to-point protocol utilizing these mechanisms is described in [3].

6 Conclusion

Communications performance of practical systems depends first of all on available packaging technology and second on protocol considerations. No topology considered here has both scalable cost and performance, so the topology chosen must be in the context of the number of processors targetted. For a thousand processors or so, given the assumptions on mid-1990's technology discussed earlier, the grid (or torus) seems an appropriate choice. The performance of the grid will depend on the signalling protocol and will be best predicted through application simulations detailed enough to relect design decisions made at that level.

References

[1] G. Bilardi, M. Pracchi, and F. P. Preparata. A critique and an appraisal of VLSI models of computation. In H. T. Kung, B. Sproul, and G. Steele, editors, *VLSI Systems and Computations*, pages 81–88, Computer Science Press, Inc., Rockville, MD, 1981.

[2] G. Brebner. Relating routing graphs and two-dimensional grids. In P. Bertolazzi and F. Luccio, editors, *VLSI: Algorithms and Architectures*, pages 221–231, Elsevier Science Publishers B.V., Amsterdam, 1985.

[3] G. T. Byrd, R. Nakano, and B. A. Delagi. *A Point-to-point Multicast Communications Protocol.* Technical Report KSL-87-02, Knowledge Systems Laboratory, Stanford University, January 1987.

[4] D. W. Dobberpuhl, R. M. Supnik, and R. T. Witek. The MicroVAX 78032 chip, a 32-bit microprocessor. *Digital Technical Journal*, (2):12–23, March 1986.

[5] Capt. B. J. Donlan, J. F. McDonald, R. H. Steinvorth, M. K. Dodhi, G. F. Taylor, and A. S. Bergendahl. The wafer transmission module. *VLSI Systems Design*, 7(1) 54–58, 88–90, January 1936.

[6] Electronic News, July 1,1985.

[7] C. K. Lau, et. al. A high performance half-micron gate CMOS process for VLSI. In *Proceedings of the 1985 International Conference on Computer Design: VLSI in Computers*, IEEE, October 1985.

[8] T Feng. A survey of interconnection networks. *Computer*, 12–27, December 1981

[9] C. W. Ho, D. A. Chance, C. H. Bajorek, and R. E. Acosta. The thin-film module as a high-performance semiconductor package. *IBM Journal of Research and Development*, 26(3) 286–296, May 1982.

[10] D. H. Lawrie Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, December 1975.

[11] C E. Leiserson. *Area-Efficient Graph Layouts (for VLSI).* Technical Report CMU-CS-80-138, Carnegie-Mellon University, August 1980

[12] C. E. Leiserson. Fat-trees: univeral networks for hardware-efficient supercomputing. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 393–402, IEEE, 1985.

[13] C. Mead and M. Rem. Minimum propagation delays in VLSI. In *Caltech Conference on VLSI*, pages 433–439, January 1981.

[14] D. Nelsen. Personal Communication.

[15] F. P. Preparata and J. Vuillemin. The cube-connected cycles: a versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[16] D. A. Reed and H. D. Schwetman. Cost-Performance bounds for multimicrocomputer networks. *IEEE Transactions on Computers*, C-32(1):83–95, January 1983.

[17] C. L. Seitz. Ensemble architectures for VLSI—a survey and taxonomy. In *1982 Conference on Advanced Research in VLSI*, MIT, January 1982.

[18] C. L. Seitz. Experiments with VLSI ensemble machines. *Journal of VLSI and Computer Science*, 1(3), 1984.

[19] C L. Seitz. Self-timed VLSI systems. In *Caltech Conference on VLSI*, pages 345–355, January 1979.

Topology	Number of Ports	Longest Path	Concurrency
Completely connected	$O(n^2)$	$O(1)$	$O(n^2)$
Crossbar	$O(n^2)$*	$O(1)$	$O(n)$
Banyan	$O(n \log n)$	$O(\log n)$	$O(n)$
Boolean k-cube ($n = 2^k$)	$O(n \log n)$	$O(\log n)$	$O(n)$

*The number of links is $O(n)$

Table 1: Scalable Concurrency Topologies. [n = # processors]

Topology	Number of Ports	Longest Path	Concurrency	Area
Ring	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Global bus	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Perfect shuffle	$O(n)$	$O(\log n)$	$O(\frac{n}{\log n})$	$O(\frac{n^2}{\log^2 n})$
Cube-connected cycles	$O(n)$	$O(\log n)$	$O(\frac{n}{\log n})$	$O(\frac{n^2}{\log^2 n})$
Binary tree	$O(n)$	$O(\log n)$	$O(\frac{n}{\log n})$	$O(n)$
4-4 Trees	$O(n)$	$O(\log n)$	$O(\frac{n}{\log n})$	$O(n)$

Table 2. Scalable Cost Topologies [n = # processors]

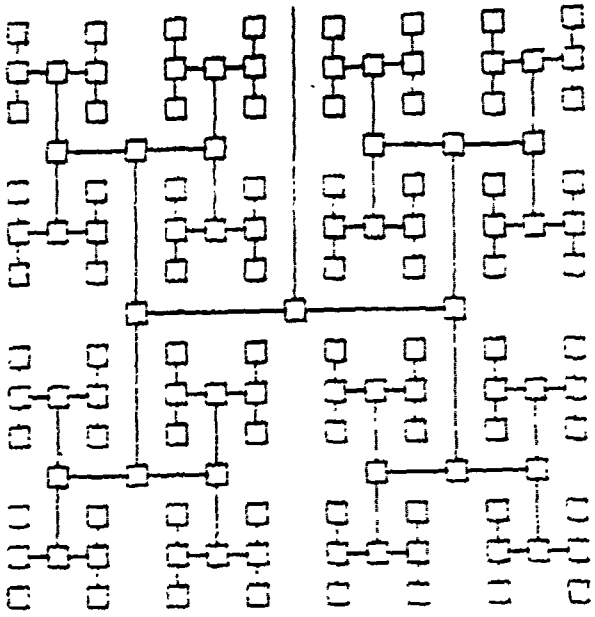


Figure 1: Recursive-H binary tree.

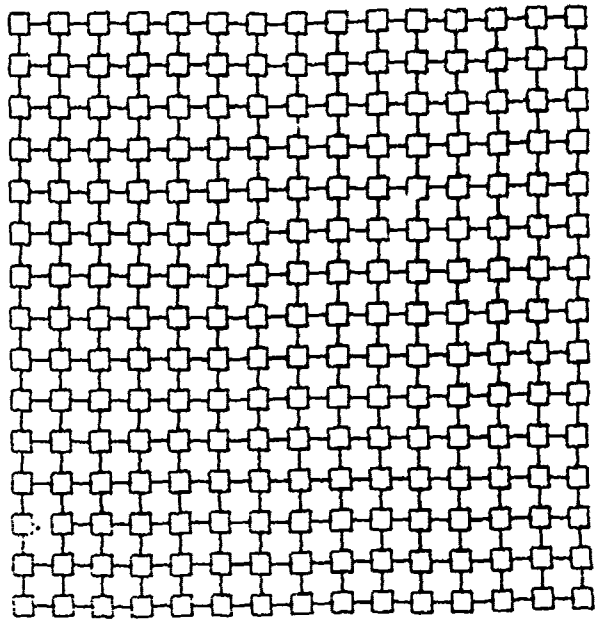
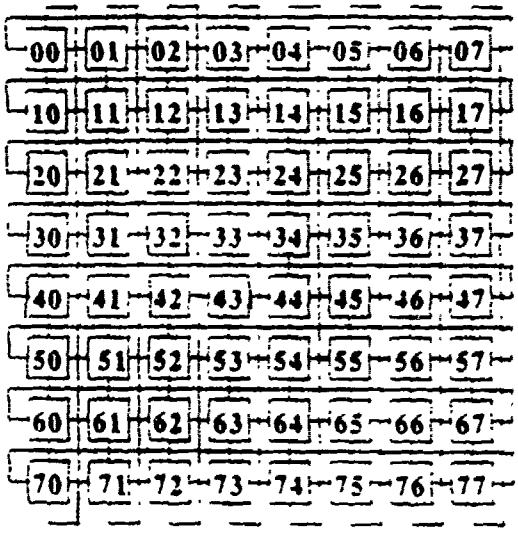
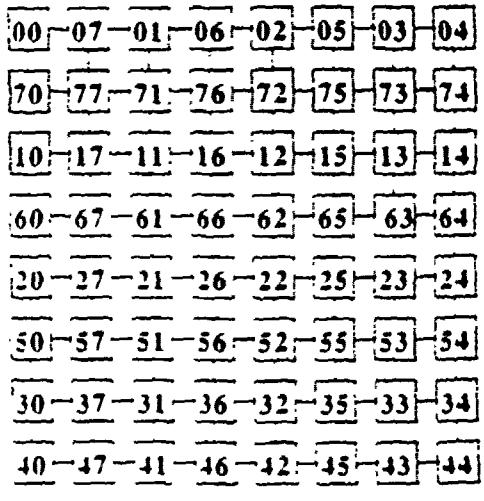


Figure 2: Two-dimensional grid.



(a)



(b)

Figure 3 Torus (a) and renumbered grid (b)

Knowledge Systems Laboratory
Report No. KSL-87-44

August 1987

A Dynamic, Cut-Through Communications Protocol with Multicast

Gregory T. Byrd, Russell Nakano, and Bruce A. Delagi

KNOWLEDGE SYTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, CA 94305

A Dynamic, Cut-Through Communications Protocol with Multicast*

Greg Byrd†
Department of Electrical Engineering
Stanford University
Stanford, CA 94305

Russell Nakano‡
Department of Computer Science
Stanford University
Stanford, CA 94035

Bruce A. Delagi
Worksystems Engineering Group
Digital Equipment Corporation
Maynard, MA 01754

*This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875.

†Supported by an National Science Foundation Graduate Fellowship, with additional support provided by the Dept. of Electrical Engineering. Any opinions findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

‡Author's present address: Digital Equipment Corporation, 100 Hamilton Avenue UCO-1, Palo Alto, CA 94301.

Abstract

This paper describes a protocol to support point-to-point interprocessor communications with multicast. Dynamic, cut-through routing with local flow control is used to provide a high-throughput, low-latency communications path between processors. In addition, multicast transmissions are available, in which copies of a packet are sent to multiple destinations using common resources as much as possible. Special packet terminators and selective buffering are introduced to avoid deadlock during multicasts. A simulated implementation of the protocol is also described.

1 Introduction

This is a revision of an earlier paper [1], in which we presented a high-performance point-to-point communications protocol with multicast capabilities. The protocol described here is essentially the same, but an effort has been made to describe the protocol in terms that more closely correspond to the intended hardware implementation.

The protocol described in this paper is designed to effectively utilize network resources. Dynamic, cut-through routing with local flow control is used to provide a high-throughput, low-latency communications path between processors. In addition, a multicast facility is provided, in which copies of a packet are sent to multiple destinations, using common resources as much as possible.

Dynamic routing means that the communications channel to be used is chosen at transmission time, based on what channels are then available. The alternative, static routing, would prescribe a specific channel for every destination—if that channel were not available, the transmission would be blocked. Dynamic routing, by adapting to current channel usage, attempts to balance the network load. It is especially useful when the communications traffic is unpredictable or variable over time [2]. Balancing the load allows more of the communications resources of the system to be well used throughout a computation.

Cut-through routing means that a routing decision is made on the fly, as a packet is received, rather than after buffering the entire packet. For example, in "virtual cut-through" routing [3], the packet is passed on a word at a time, until a desired channel is blocked, at which time the packet is buffered.[1] "Wormhole" routing [5], on the other hand, uses flow control signals to halt the packet flow, rather than buffering it. Cut-through routing offers reduced buffering requirements (since the packet need not be buffered at each node) and low latency. [6,7]

Flow control, in general, is any mechanism which attempts to regulate the flow of information from a sender to match the rate at which the receiver can accept it [8]. In this protocol, a transmission may be blocked and resumed in the event of network congestion. If an output channel becomes blocked, the sender stops sending data and halts the flow of data from upstream. When the channel becomes unblocked, the transmission is continued from where it was halted. The flow control mechanism is local, because actions are taken based on the state of the downstream component rather than global information about the entire network.

Multicast transmissions in a point-to-point network allow a packet to be sent to multiple destinations, using common resources as much as possible. The packet is replicated as needed, and subsets of the original target list are assigned to the copies. Thus, "virtual busses" are available precisely as and when they are needed. Selective buffering and special packet terminators allow potential

[1] A related concept is staged circuit switching, described in [4].

deadlock conditions in multicasts to be detected and avoided.

The network components which define the protocol are introduced in section 2, and the protocol itself is described in section 3. Section 4 presents a hypothetical hardware implementation of the protocol, while section 5 describes the implementation in the CARE simulation system.

2 Components

This section defines the network components used by the protocol. The protocol is defined by the behavior of these components and the values that are passed among them. Of course, these components do not necessarily correspond to distinct physical entities in a machine which implements this protocol—they are merely a useful means of specifying the communications behavior of such a machine.

The *site* component corresponds to a processor-memory pair in the target machine. In particular, a site contains an operator, an evaluator, a router, some local storage, and some network interface components, which are called net-inputs and net-outputs (see figure 1).

The *evaluator* is the part of the site which executes application code. The evaluator can request network activity, but otherwise has no role in the network behavior of the system, so very little will be said about it in this paper.

The *operator* is responsible for handling system-level activity, including communication. In the target machine, it would create packets to be sent over the network and accept transmissions destined for its associated processor. The operator and evaluator communicate through shared local memory. The details of operator-evaluator communication will not be addressed in this paper.

The site components which interface directly to the network are called *net-inputs* and *net-outputs*. On each site, there is a net-input/net-output pair connected to the operator, for local packet origination and delivery, as well as a pair for every communication channel to the network.[2] We will refer to the pair connected to the operator as the "local" net-input and net-output. Because of cut-through routing, net-inputs and net-outputs are only required to have enough storage for one word of a packet, rather than the entire packet, where a "word" is long enough to specify a target site.

The *router* connects all the net-inputs on a site to all the net-outputs. When it receives a packet from a net-input, it determines the destination (or destinations) and makes the connection to the appropriate net-output (or net-outputs). Also, flow control information from the net-outputs are relayed by the router to the appropriate net-input.

A pair of *fifo-buffers* queues packets between the operator and local net-input and net-output. The *upstream* fifo-buffer queues packets from the network to

[2] The exact number of net-input/net-output pairs required by a site depends on the network topology.

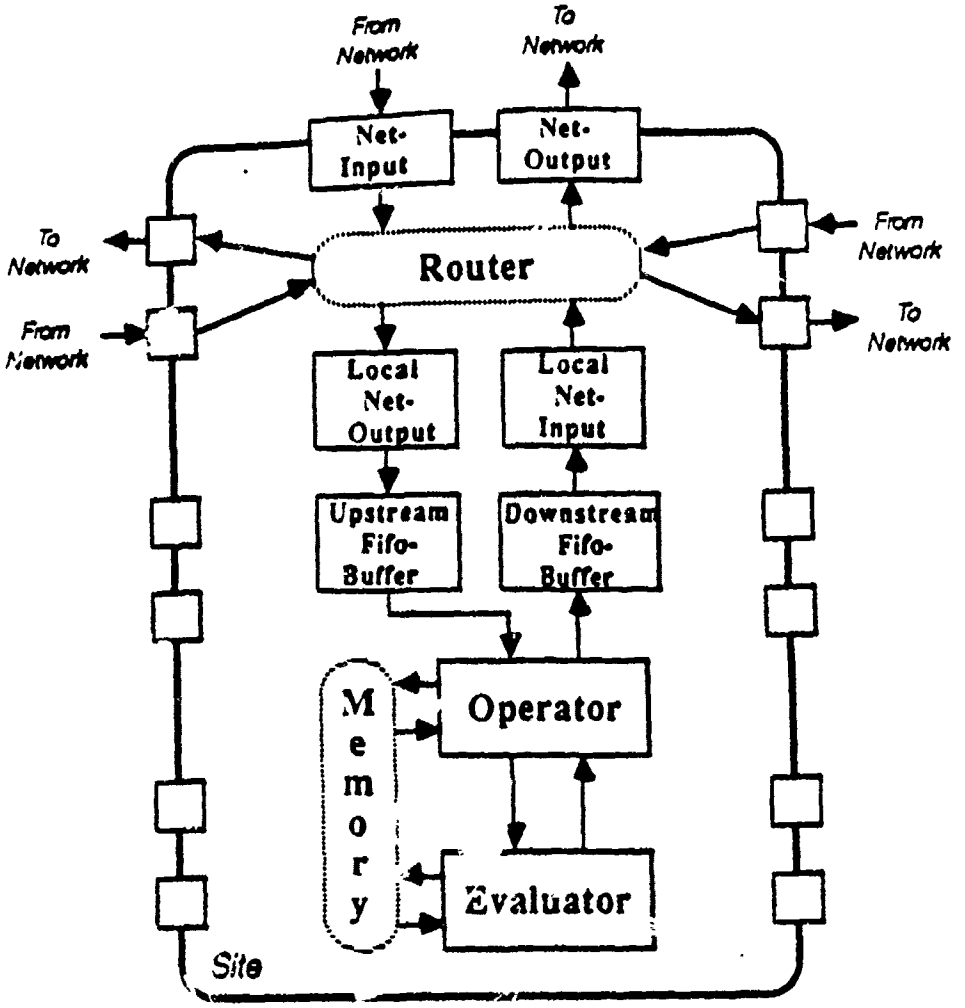


Figure 1: Components of a CARE *site*.

the operator: the *downstream* fifo-buffer queues packets from the operator to the network.

3 The Protocol

3.1 Packets

Figure 2 shows the organization of a packet. The first part of a packet is devoted to the *target entries*. Each entry specifies a target site, as well as other information that will be used when the packet arrives at the site. Following the target entries are zero or more words of *data* and a one-word *packet terminator*. The operator determines the status of a packet by examining its terminator.[3]

Each word in a packet is tagged, so that target entries may be differentiated

[3] As described in section 4.1

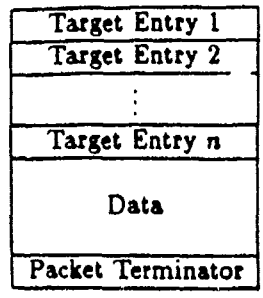


Figure 2: Organization of a packet.

from data. There are two types of tags used for specifying a target site—one which indicates that there is only one target for this packet (i.e., *unicast*), and one which indicates that there may be more than one (i.e., *multicast*). This allows the router to handle unicasts efficiently, without the extra mechanisms required for multicasts described later. There are also a tags for the other words in a target entry, which do not specify a site.

Also, tags are used to implement several special characters required for the protocol. There are two types of *pad characters*: one for denoting a null target entry, and one for indicating that there is no word available for transmission. Finally, there are three distinct packet terminators—**:end-of-packet**, **:local-end-of-packet**, and **:abort-packet**. The uses of these special characters will be further explained as the protocol is described.

Table A summarizes the tags needed to implement target entries and special characters.

Target Sites	**:unicast-site** **:multicast-site**
Pad Characters	**:null-target** **:null-transmission**
Terminators	**:end-of-packet** **:local-end-of-packet** **:abort-packet**

Table A: Tags used by communications system.

3.2 Packet Transmission

The transmission path of a packet is shown in figure 3. First, an evaluator requests a packet transmission. For the moment, assume a unicast transmission (only one target). The operator then sends the packet (through a fifo-buffer) to the local net-input. The router decides which net-output should receive the packet, based on the target site and the availability of net-outputs, sets up a connection between the local net-input and the selected net-output, and begins the transfer of the packet. Each non-local net-output is physically connected to a net-input on a (logically) neighboring site. When available, this net-input accepts the packet, and its router sends the data to the local net-output, if the target site has been reached, or to another net-output, if not. This continues until the target site has been reached, where the local net-output delivers the packet to the operator (through a fifo-buffer). The operator can then perform whatever operation is specified by the packet, such as storing the value in memory or queueing some operation for the evaluator, for example.

If the packet has more than one target, the router may split it—that is, it may send (essentially) the same packet to several net-outputs. This is called a *multicast* transmission. Each transmitted packet contains a distinct *subset* of the targets of the original packet.[4] The copying operation is done during transmission, one word at a time, as opposed to buffering the entire packet and making copies. If any branch of the multicast is blocked, the net-input sends **:null-transmission** characters down the other branches until valid data may be sent down all the paths. The pad characters (either **:null-target** or **:null-transmission**) are thrown away when received by a fifo-buffer.

3.3 Flow Control

Flow control information, in the form of status signals, flows in the direction opposite to packet transmission. There are three distinct status signals, as

[4] Each copy of the packet as it is transmitted will have the same number of target entry "slots," but some of them will contain null entries.

Status	*Meaning*
'open	Available to receive data.
'wait	Busy or network is blocked; do not send more data.
'abort-request	Potential deadlock detected.[a]

[a] Only a fifo-buffer may originate the **'abort-request** signal.

Table B: Flow-control signals.

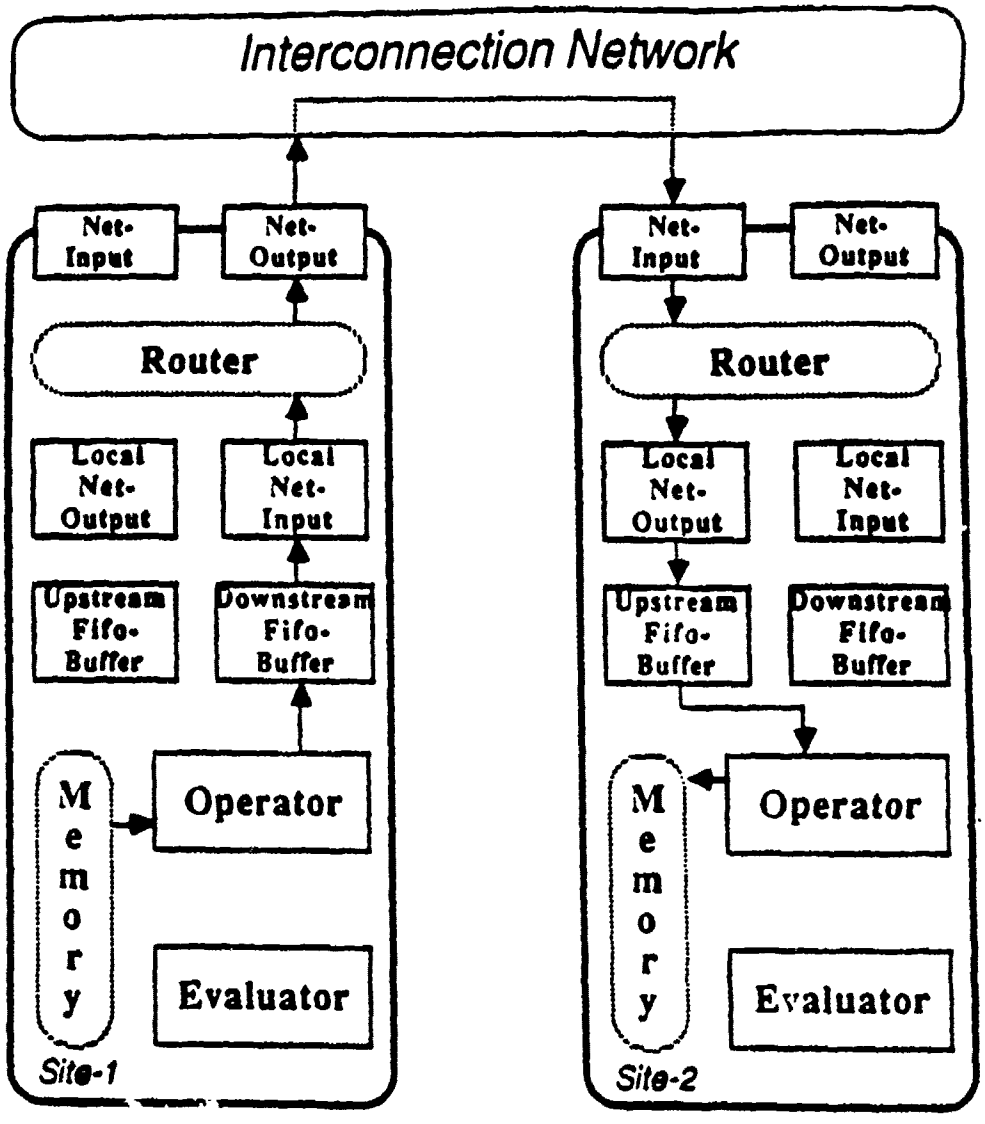


Figure 3: Network component interconnections. Packets travel in the direction marked by arrows. Flow control information flows in the opposite direction.

shown in Table B. The status signals are used to indicate to the upstream component whether data can safely be transmitted.

An 'open signal is used to indicate that the component is ready to receive the next word of the packet. If the transmission becomes blocked for some reason, a 'wait signal is sent upstream to temporarily halt the flow of data. Finally, the 'abort-request signal indicates that a potential multicast deadlock condition has been detected and the transmission may be aborted.

3.4 Deadlock Avoidance

3.4.1 Unicast Deadlocks

Dally and Seitz [5] have developed a deadlock-free unicast transmission scheme for wormhole routing, based on virtual channels. Our strategy is different—if progress cannot be made, a packet may be temporarily buffered at an intermediate site. In this way, at least one of the packets responsible for a deadlock will be removed from the network, so that the other packets may make progress. Thus, this protocol is a compromise between virtual cut-through [3], in which the packet is *always* buffered when it is blocked, and wormhole routing [5], in which the packet is *never* buffered.

More specifically, if the number of connection attempts for an acceptable net-output exceeds a threshold, then the local net-output is considered as a potential target. If the local net-output becomes available before the desired net-output, the packet is buffered, freeing its upstream channels. When the operator examines the packet and discovers that the packet was targetted for another site, it will retransmit the packet. Assuming packets cannot be infinitely long, either the local net-output or an acceptable remote net-output will eventually become free, so that deadlocks can be avoided, as long as there is sufficient space in the site at the front edge of the transmission.

3.4.2 Multicast Deadlocks

The existence of packet multicasts introduces the possibility of another type of deadlock. A packet traveling through the network acquires the use of network resources (e.g., net-inputs and net-outputs) and simultaneously excludes the use of those resources by other packets. Without special attention paid to the possibility of deadlocks, it is possible that resources are consumed to perform the multicast, but completion of the transmission is not possible because the resources acquired are insufficient.

Figure 4 illustrates an example of how multicast deadlock can arise. Suppose we have two multicast transmissions, call them A and B, with common destinations, *site-1* and *site-2*. Suppose that one of the packets from multicast A has already gained access to the local net-output on *site-1*. A packet from multicast B has similarly gained access to the local net-output on *site-2*. For

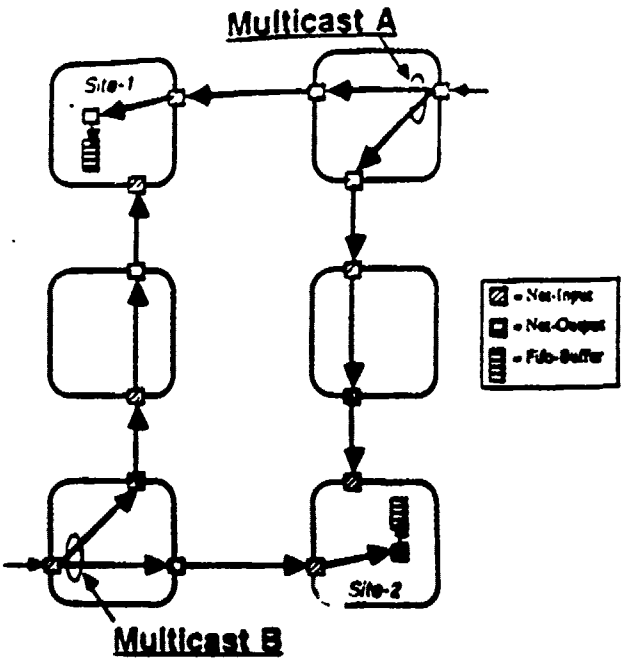


Figure 4: Example of deadlock in a multicast.

multicast *A* to continue, it needs to gain access to the local net-output of *site-2*;[5] for *B* to complete, it needs to gain access to the local net-output on *site-1*. Also, neither of the multicasts can release the resources it has already required until the transmission is completed. Since each multicast has acquired a resource that the other needs, a deadlock results.

In order to recover from such a situation, the system must:

- Detect a potential deadlock condition, such as the situation described above;
- Back out of the unsafe condition (by aborting one or more transmissions, thereby releasing some set of resources); and
- Retransmit the aborted packets later, when the network is (hopefully) less congested.

Whenever a packet is split for multicast, the protocol requires that a copy of the original packet (with a complete target list) be sent to the local net-output. This packet will then be stored in a fifo-buffer, so that it may be retransmitted in the case that the current multicast must be aborted due to deadlock.

[5] The transmission cannot continue because the net-input cannot send any words until all branches of the multicast are ready to receive it. Since the branch waiting for the local net-output of *site-2* is blocked, none of the branches may proceed.

A potential deadlock is detected by the fifo-buffer when the number of consecutive :null-transmission characters exceeds a threshold. This indicates that one or more branches of the multicast have been blocked for a long time, which implies the possibility of deadlock. When the threshold is exceeded, the fifo-buffer asserts an 'abort-request signal upstream, so that the router may abort the transmission if necessary.

A multicast is aborted by sending the :abort-packet terminator downstream—all operators which receive a packet with this terminator will ignore the packet. Also, the operator which receives the copy of the original packet can tell whether it needs to be retransmitted by looking at its terminator.

These actions are sufficient to prevent persistent deadlock during multicasts. However, since there is finite storage in the system, a scenario can be constructed in which all the storage becomes committed and no packets can be delivered. The protocol does not prevent this type of resource exhaustion. The assumption is made that the designed capacity of the system is sufficient for its applications.

4 Implementation

This section provides a detailed description of the behavior of each of the network components in a hypothetical hardware implementation. Figure 5 shows a "generic" network component, with its input and output ports. The input and output ports are used to pass packets and flow control information—packets flow downstream, flow control signals flow upstream. The packet-in port accepts data from upstream, and the packet-out port sends data downstream; the status-in port accepts flow control signals from downstream, and the status-out port sends flow control signals upstream.

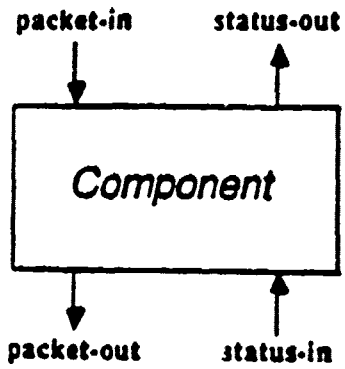


Figure 5: Generic network component.

4.1 Operator

The operator sends and receives packets through the network and through the memory it shares with the evaluator. Thus, it has more than one set of ports for packet communication. To avoid confusion, the ports it uses to communicate with the network are prefixed **network-** (e.g., **network-packet-in**), while the ports used for communication with the evaluator are prefixed **evaluator-** (e.g., **evaluator-packet-in**). Only network communication will be discussed in this paper.

With respect to the network, both the upstream and downstream components of an operator are fifo-buffers. The upstream fifo-buffer queues packets from the local net-output and sends them to the operator. The downstream fifo-buffer queues packets from the operator and sends them to the local net-input.

4.1.1 Sending a Packet

The operator has a queue of *operations*, or requests, which it services in order of arrival. If the head of this queue is a packet to be sent out into the network, and **network-status-in** is **'open**, indicating that the downstream fifo-buffer is ready to accept a packet, the operator sends the packet (with an **:end-of-packet** terminator) through the **network-packet-out** port.

4.1.2 Receiving a Packet

A packet arrival at the operator is signalled by the appearance of a target entry word on the **network-packet-in** port. The **network-status-out** port is set to **'open**, which signals the upstream fifo-buffer to keep sending packet words, and the packet is stored in a temporary buffer.

The action taken by the operator when the packet is completely received depends on the type of packet terminator. There are three types of terminators, shown in Table C, and their interpretations are given below.

The arrival of an **:end-of-packet** signifies that the packet transmission was successful. The operator sends **'wait** to the upstream fifo-buffer (through **'network-status-out**) until the packet is serviced (e.g., an evaluator operation

Terminator	*Meaning*
:end-of-packet	Normal packet termination.
:abort-packet	Packet is to be discarded by operator.
:local-end-of-packet	Treat as **:end-of-packet**, except ignore all packet targets other than the local site.

Table C: Packet terminators.

is queued). When the operator is ready to receive the next packet, it asserts **'open**.

If the operator notices that some or all of the target addresses of the received packet do not correspond to its own address, the packet is sent back out into the network.[6] This might happen for one of the following reasons:

1. During a unicast transmission, a net-input could not make a connection to the desired net-output. The packet is forced into the local fifo-buffer, so that the operator may resume the transmission at a later time, freeing up the net-input and its upstream components.

2. A multicast transmission (originated locally) was aborted. The local fifo-buffer received a copy of the packet with a complete target list, so that the packet could be retransmitted in case of an abort.

A **:local-end-of-packet** terminator instructs the operator to accept the packet, as in the case of **:end-of-packet**, but to ignore any non-local target addresses (i.e., no retransmission). This indicates that a multicast was successful and does not have to be retried.

The arrival of an **:abort-packet** terminator instructs the operator to ignore the packet. In other words, the temporary buffer holding the packet is released without servicing the packet.

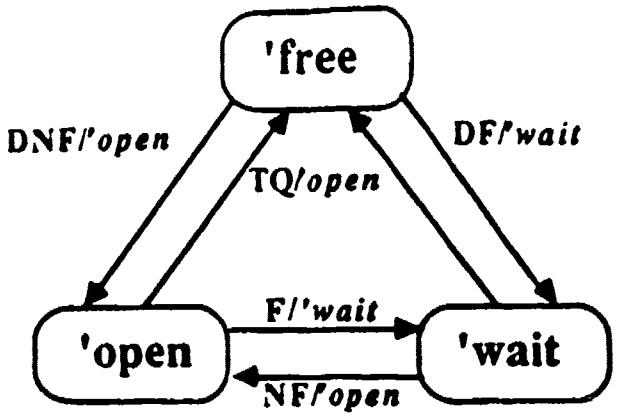
4.2 Fifo-buffer

Each site has two fifo-buffers, which have identical behavior but perform slightly different functions. One fifo-buffer is upstream with respect to the operator, and the other is downstream. The fifo-buffer can be thought of as three distinct parts: the *input*, the *queue*, and the *output*.

The queue is a simple FIFO queue, with one-word input and output ports. It responds to a **'take** signal from the output by placing the oldest item in the queue on the output ports. It responds to a **'put** signal from the input by placing the incoming data at the tail of the queue. It also presents a *queue-status* signal to both the input and output, which can be **'empty**, **'some**, or **'full**. If the queue is empty, it sends a pad character to the output in response to a **'take** signal.

On its output side, the upstream fifo-buffer is connected to the operator, while the downstream fifo-buffer is connected to the local net-input. The output interprets an **'open** signal on **status-in** by sending **'take** to the queue and sending the resulting output downstream. Nothing is removed from the queue if **status-in** is **'wait**.

[6] If any of the targets are local, the operator keeps a copy of the packet and strips the local targets from the retransmitted packet.



Condition	Meaning
DF	Data arrives, and queue full.
DNF	Data arrives, and queue not full.
F	Queue full.
NF	Queue not full.
TQ	Terminator queued.

Figure 6: Fifo-buffer state diagram.

On its input side, the upstream fifo-buffer is connected to the local net-output, and the downstream fifo-buffer is connected to the operator. The fifo-buffer needs to keep track of whether the terminator for the current packet has arrived, because of the multicast abort procedure needed for deadlock avoidance, so we describe the input handler as a finite state machine, whose state diagram is shown in figure 6. The labels on the arcs represent the condition which caused the transition and the status signal asserted on status-out as a result.

The fifo-buffer input begins in the 'free state. Whenever new data arrives on the packet-in port, if the queue is not full, the 'open state is entered and 'open is asserted on status-out. If the queue is full, the 'wait state is entered and 'wait is asserted; when space becomes available in the queue, the 'open state is entered and 'open is asserted. If the queue becomes full at any point in the transmission, the 'wait state is entered and the 'wait signal is asserted on status-out, so that no more data will be sent from upstream. When space becomes available, the 'open state is re-entered, and 'open is sent upstream to

resume the flow of data.

When the fifo-buffer is in the 'open state, a "time-out" may occur, which indicates that number of consecutive :null-transmission characters has exceeded a threshold. When this happens, it remains in the 'open state and asserts 'abort-request on the status-out port.

When a packet terminator arrives, if the queue is not full, the 'free state is entered and 'open is asserted on status-out. If the queue is full, the 'wait state is entered first, which asserts 'wait until space is available in the queue. Then the 'free state may be entered. At this point, the fifo-buffer is ready to receive the next packet.

4.3 Net-Input

The downstream component from a net-input is a router, but the values on the status-in port are actually originated from a downstream net-output and are passed through the router. If the net-input is local (connected to an operator), its upstream component is a fifo-buffer; otherwise, its upstream component is a net-output (on a neighboring site).

The net-input serves as a one-word data buffer and relays flow control information to its upstream component. It has a two-phase operation:

1. During phase one, the status latch is opened, and the current value of status-in flows upstream. This value will either be 'open or 'wait—the router will not allow an 'abort-request signal to ever reach the net-input. The data latch (fed by packet-in) is closed during this phase, and the stored value is output on packet-out.

2. During phase two, the net-input closes the status latch and examines the latched signal. If the signal is 'open, it opens the data latch, allowing data to flow downstream. If the signal is 'wait, the data latch remains closed. In any case, the data latch is closed at the end of this phase.

4.4 Net-Output

The upstream component of a net-output is always a net-input. On the downstream side, the local net-output is connected to the fifo-buffer which delivers packets to the operator, while a non-local net-output is connected to a net-input on a logically neighboring site.

The operation of the net-output is the same as the net-input, except that the phases are reversed. The net-output conditionally latches data during phase one, and allows flow control signals to flow upstream during phase two. The only other difference is that the 'abort-request signal may be passed upstream.

Table D summarizes the net-input and net-output operations during the two communication phases.

Component	Phase One	Phase Two
Net-Input	Open status latch to allow status information to flow upstream.	Latch status from downstream and conditionally open data latch to allow data to flow downstream.
Net-Output	Latch status from downstream and conditionally open data latch to allow data to flow downstream.	Open status latch to allow status information to flow upstream.

Table D: Communication cycle phases.

4.5 Router

The router connects the net-inputs and net-outputs of a site, and is responsible for:

- Determining to which net-outputs a packet should be sent, based on the packet's target addresses, the system routing strategy, and the current availability of net-outputs;
- Creating, maintaining, and deleting the connections between a net-inputs and sets of net-outputs, including transmitting data and flow control signals between them; and
- Sending appropriate pads and packet terminators, in order to implement the deadlock avoidance mechanism.

For a unicast transmission, the function of the router is quite simple. Upon examining the packet target, it selects a net-output (possibly the local one) to continue the transmission, based on the location of the target site relative to its own and on the availability of net-outputs. If no connection can be made, a 'wait signal is sent to the requesting net-input until a net-output becomes available. After a net-output is selected, the router maintains the connection by sending data from the net-input to the net-output and sending flow control signals from the net-output to the net-input. When the packet transmission is completed, the net-output becomes available to accept another connection.

During a multicast transmission, the packet targets are read one at a time, and the connections to net-outputs are made as the targets are read. For each

net-input the router keeps track of the type of its current connection. There are three possible connecton types:

'unicast The packet is being transmitted to only one target, either because there was only a single target in the packet, or because the packet is being "passed through" because the local net-output was not available.

'all-remote The packet has multiple sites in its target list, and the router has made connections to multiple net-outputs. The packet's target list contained only non-local sites.

'some-local The packet has multiple sites in its target list, and the router has made connections to multiple net-outputs. The packet's target list included the local site.

In the next two sections, we present further details about how connections are made and how multicasts are handled.

4.5.1 Making a Connection

Making a connection involves determining the logical "direction" (e.g., up or down) of the target from the local site, then determining which net-output should be used for that direction, and finally updating the connection tables and starting the packet transmission.

Determining the logical direction depends on the network topology and is usually straightforward. For example, a grid or torus requires only some arithmetic comparisons between the target address and the local address to get Up, Down, Right, Left, or some combination of these. A hypercube, on the other hand, requires an exclusive-OR operation to see which bits in the destination address are different than the local address. Equally simple operations can be envisioned for most other network topologies, as well.

The protocol does not prescribe a particular routing policy for the network. Instead, information about possible connections is "hard-wired" into the router in the form of a priority network. Conceptually, we model the priority network as a *preference table*—for every logical direction, we provide a prioritized list of net-outputs that may be considered. Examples of routing strategies which may be implemented in this way are (1) try all net-outputs, starting with the closest to the target, (2) try only one net-output (static routing), and so forth.

Given a direction, the router checks the status of each net-output in the preference table, in turn, until an available net-output is found. If none is available, then the connection fails, and **'wait** is sent upstream to the net-input.

4.5.2 Multicast Transmissions

When a multicast packet arrives, the router makes a connection for each packet target, one at a time. If the connection for a target has already been made (in response to an earlier target), the target entry is merely transmitted downstream

to that net-output. Whenever a target entry is transmitted, :null-target characters are sent down all of the other connections. In this way, the target list is partitioned along several paths. When the packet data is received by the router, it is transmitted to all the connected net-outputs. If any of the downstream paths becomes blocked, :null-transmission characters are transmitted down all the other paths.

There is an additional complication for the router, however, since the local net-output must be sent a copy of the packet to be buffered, in case the transmission is aborted and must be retried. Because of the special :unicast-site tag, the router knows immediately whether a packet should be treated as a multicast or unicast. Note, however, that since the router only looks at one address at a time, the router cannot determine when the *last* target occurs for a particular branch of the multicast. Thus downstream routers may mistakenly interpret a packet with only one target as a multicast. As a result, unnecessary local copies of this packet will be made as it makes its way to its target site.[7]

When the first target of a multicast is received, the router tries to connect to the local net-output, as well as the net-output specified by the preference table. If the local net-output is not available, then the packet is not split at this site. Instead, the entire packet is sent down the remote connection. In this way, the packet will either sequentially visit each target on the list or will finally reach a site where it may be split.

If at any time during the connection process, a desired net-output is not available, a 'wait is sent upstream to the net-input to halt the flow of additional targets. While waiting for a net-output to become free, the router must send target pad characters down the established connections. Unlike in the unicast case, we cannot decide to divert this target to the local net-output, since then there would be no way to tell which targets were actually serviced and which were diverted. Therefore, to avoid the possibility of deadlock during target processing, the *local* net-output must be sent *data* pad characters, so that the downstream fifo-buffer can time out, if appropriate, and the multicast can be aborted.

If the transmission completes successfully (i.e., is not aborted), the received packet terminator is passed on to all the remote (non-local) net-outputs, but the local net-output may be sent a modified terminator, as follows. If the received terminator is :abort-packet, it is sent as is, instructing the local operator to ignore the packet. If the received terminator is :end-of-packet, the terminator sent to the local net-output depends on the connection type:

'all-remote An :abort-packet is sent, since the packet should not be retransmitted and may be ignored.

[7] The router could be optimized to notice when an 'all-remote connection only uses a single connection— an :abort-packet could then be sent to the local fifo-buffer, since there is no possibility of deadlock and thus no retransmission will be necessary.

'some-local: A **:local-end-of-packet** is sent, instructing the operator to accept the packet for the local targets, but to ignore the remote targets (i.e., do not retransmit).

If, during the multicast transmission, the router receives an **'abort-request** signal from the local net-output (generated by the downstream fifo-buffer), the router aborts all the remote connections for the connected net-input by forcing the net-outputs to latch an **:abort-packet** terminator. An **'open** signal is passed upstream to the net-input, and the transmission proceeds as if it were a unicast transmission destined for the local operator. When the packet terminator is received, it is passed directly to the local net-output. Note that an **:end-of-packet** will cause the packet to be retransmitted by the operator,[8] since there are non-local targets, and an **:abort-packet** will cause the packet to be discarded.

5 CARE Implementation

In this section, we provide an overview of the implementation of the protocol in the CARE simulation system. CARE is a library of functional modules and instrumentation built on top of an event-driven simulator [9], which is used to investigate parallel architectures. The typical CARE architecture is a set of processor-memory pairs (*sites*) connected by some communications network, though it can also be configured to represent a system of processors communicating through shared memory. The behavior and relative performance of CARE modules can easily be changed, and the instrumentation is flexible and useful in evaluating the performance of an architecture or in observing the execution of a parallel program.

CARE is implemented using Flavors—an object-oriented extension of Zetalisp [10]. Roughly speaking, each component described in section 2 is implemented as an object (an *instance* of a flavor). (One notable exception is the router—its functions and tables are assumed by the *site* object, rather than implemented as a separate component. Also, the memory at a site is not explicitly represented as an object, but exists implicitly in the simulator.) Associated with each object is a set of *instance variables*, used to hold state information, and a set of *methods*, procedures used by the object to respond to messages from other objects.[9] The instance variables loosely correspond to the ports and state variables used to describe the protocol in section 3. In particular, each of the components which are described in terms of a state machine has a instance variable, **packet-status**, which hold the current state of the component.

[8] If there are local targets, a copy of the packet will be kept and the local targets will be removed from the target list upon retransmission.

[9] Objects and messages are only a software tool used by the simulator. Sending *messages* between objects in the simulator has no particular correspondence to sending *packets* between components in the target machine.

These objects communicate through shared structures called *vias*, which represent unidirectional data paths. These are the "wires" which connect the components' "ports." Asserting a value on the sending end of the via immediately (in simulated time) triggers an event for the object at the other end. Therefore, a via can be considered a zero-delay wire which can transmit any arbitrary value (not just single bits).

The simulation is functional,[10] rather than circuit-level, and event-driven, rather than clock-driven, because cycle-by-cycle simulation of a parallel machine would be extremely time-consuming, especially when the number of processors is large. For this same reason, we do not wish to model the transmission of a packet one word at a time. Instead, a packet is represented by two distinct parts, one representing the contents of the packet, and the other representing the packet terminator. In the following discussion, *packet* will refer to the first part (representing the front edge of a "real" packet), and *packet terminator* will refer to the terminator part.

In the simulation environment, explicit packet terminators allow us to (1) implement the deadlock avoidance mechanisms described earlier, and (2) model the transmission of a packet through the network in terms of its front edge and its back edge. The transmission time of a packet is the time between arrival of its front edge and its terminator. In this way, we can accurately model the transmission of the packet without explicitly representing every word.

In the following subsections, we describe how the protocol is implemented in terms of objects, packets, and packet terminators.

5.1 Operator

The time required to transfer a packet from the operator to a fifo-buffer (one word at a time) would be proportional to the size of the packet. To model this, the operator delays an appropriate time between sending a packet and sending its terminator. When the transmission time of the packet has elapsed, the terminator is sent as soon as an 'open signal is received from the fifo-buffer. This is a simplified model, since there can be arbitrary delays involved in freeing up space in a full buffer, but the fifo-buffer output module ensures that the proper space is inserted between packet and terminator in the network.

A CARE operator receives a packet as described in the protocol. Note that the time between receiving the packet and its terminator is dependent on the size of the packet plus any delays encountered on its transmission path.

[10] The simulation is functional, in the sense that not every aspect of the hardware is simulated in detail. Some aspects are simulated by register transfer level behavior, while other aspects have only a functional description. For example, the communications system is simulated in terms of register transfers, while the execution of (uniprocessor) application code by the evaluator is not simulated at all—it is directly executed by the host machine. However, timing information for the execution of application code, based on measurements and estimates, is used to assure that the simulation is reasonably faithful to the execution of a "real" machine.

5.2 Fifo-buffer

In the simulator, the amount of storage in the fifo-buffer may be set at run time.[11] Each packet or packet terminator takes up one space in the buffer, no matter what its actual size.

Since we do not simulate each word of a packet transmission, the fifo-buffer cannot count pad characters to detect a potential multicast deadlock. Instead, the simulated fifo-buffer uses a time-out procedure: when the packet is received, the fifo-buffer schedules a wake-up event at random time in the future, based on the packet size (for example, between 1.5 and 3 times the packet transit time). If the packet terminator has not arrived by that time, the fifo-buffer asserts **'abort-request**. This is not a viable option for actual implementation, since a real packet header contains no information about the packet size.

On its output side, the simulated fifo-buffer is more complex than the protocol indicates. If a packet is being output from the queue, the fifo-buffer must introduce a delay between the packet and its terminator to model the packet transit time. However, the transit time is not merely proportional to packet size, because downstream blocking could cause arbitrary delays in the transmission.

The simulated fifo-buffer output transitions are shown in figure 7. In this case, the transitions are labelled with conditions and *actions*, rather than flow control signals. Some additional instance variables for the fifo-buffer are required to implement the output function. They are:

1. **transmission-status:** State of packet output.
2. **delay:** Accumulated time spent waiting.
3. **last-wait:** Event time when last **'wait** was received.

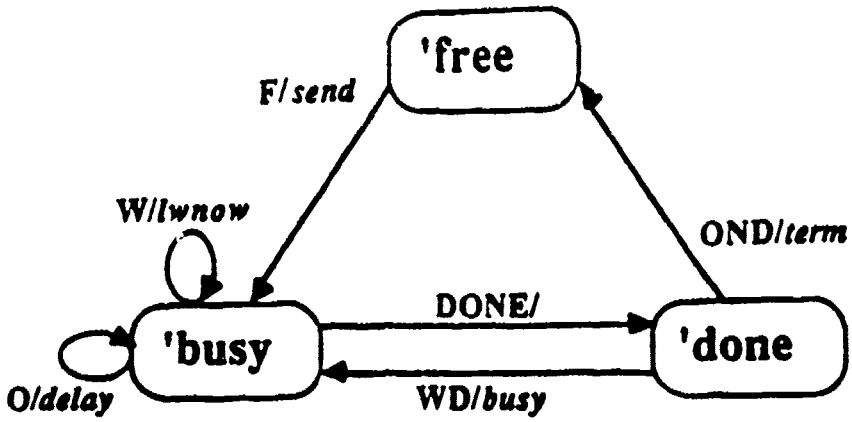
Initially, **transmission-status** is **'free**. If the downstream component requests data (**status-in** goes to **'open**) and the queue is not empty, the top of the queue, which must be a packet, is placed on the **packet-out** via, **delay** is set to zero, and **transmission-status** goes to **'busy**. Also, **transmission-status** is scheduled to go to **'done** at a time that is proportional to packet size.

If no **'wait** signals are received from downstream while the transmission is **'busy**, then the transmission will be done after the packet transit time has elapsed, and the packet terminator will be sent as soon as the downstream component is ready to receive it.

However, if **'wait** is received during **'busy**, **last-wait** is set to the current time and **waiting** is set to **t**. If **'open** is received during **'busy**, the time spent waiting is added to **delay** and **waiting** is set to **nil**.

If **'open** is received when **transmission-status** is **'done**, and **delay** is non-zero, then **'busy** is entered again, **'done** is scheduled for the current time

[11] By setting the care:***buffer-size*** variable to any positive integer, or to nil, which means "unbounded."



Condition	Meaning
F	'Free rec'd on status-in.
W	'Wait rec'd on status-in.
O	'Open rec'd on status-in.
DONE	'Done event.
WD	'Wait rec'd and [delay nonzero OR last-wait non-nil].
OND	'Open rec'd and [delay = 0 AND last-wait = nil].

Action	Meaning
send	Send packet, schedule 'done for now + transmission-time.
lwnow	Last-wait = now.
delay	Delay = delay + (now - last-wait); Last-wait = nil.
busy	Schedule 'done for now + delay; Last-wait = nil.
term	Send terminator.

Figure 7: Simulated fifo-buffer output state diagram.

plus the accumulated delay, **waiting** is set to **nil**, and **delay** is set to zero. Alternatively, if **waiting** is **t** and **delay** is zero, then **'done** has occurred in the middle of a wait; **'busy** is entered, **waiting** is set to **nil**, and **'done** is scheduled for the current time plus the difference between now and **last-wait**.

Finally, when **transmission-status** is **'done**, **delay** is zero, and **waiting** is **nil**, the top item of the queue (which must be a packet terminator) will be sent. Then **transmission-status** becomes **'free**, and the fifo-buffer is ready to respond to the next data request.

All of this is to ensure that the time between the packet and its terminator is dependent on the packet size plus any network delays along its path. The other components, net-inputs and net-outputs, do not require this added complexity on the output side. Since they merely pass packets and terminators from one point to the next,[12] the flow control signals ensure that they will maintain the proper separation between a packet and its terminator.

5.3 Net-Input, Net-Output, and Router

As mentioned earlier, the router is not an explicit object in the simulation. Instead, the *site* object performs its operations. Net-inputs and net-outputs communicate with it by passing messages (in the Flavors sense) rather than making assertions on vias. Likewise, the site updates net-input and net-output "ports" by setting instance variables.

To connect to net-outputs, the net-input sends a **:connect** message to the site, which then attempts to make the appropriate connections. The result is stored in the **connection** instance variable of the net-input. If no connection could be made, **'seek** is returned; otherwise, the type of connection (unicast, all-remote, or some-local) is returned. If only some of the desired connections could be made, the unsuccessful targets are placed in the **pending-connections** instance variable. The net-input keeps sending **:connect** messages to the site until all the targets are satisfied.

Other site methods used by the net-input include **:disconnect-remote**, which releases the connections to all net-outputs except the local one, and **:send-all**, which transmits a packet or terminator to all connected net-outputs. (**:Send-local** and **:send-remote** transmit to a subset of connected net-outputs.)

Similarly, the net-output uses the **:wait**, **:open**, and **:abort-request** methods to relay flow control signals to the site, which then makes the appropriate assertions to the connected net-input.

In the router, the **:find-direction** method determines the logical direction of a target, given its address. This is defined as a method, rather than a function, because this operation is topology-dependent. In Flavors, we can define

[12] This is in contrast to the fifo-buffer, which must *insert* the packet and terminator into the network at the proper time.

a specialized *site* object for a particular topology by changing this one method and inheriting the remaining behavior from the generic site definition.

The **setup-targets** function examines the target list, makes the connections, and copies the packet, as needed. Finally, the **make-connections** function is responsible for actually setting up connections and sending the packet downstream.

5.4 Results

Variants of this protocol have been used for many CARE simulations over the course of several months. Though the performance has not been extensively measured, the protocol appears to offer reasonable performance over a range of network loads. Deadlocks and lost packets do not occur, even when the network is extremely congested. Thus, our experience with the protocol indicates that it offers efficient and robust one-to-one and one-to-many interprocessor communication.

6 Conclusion

A protocol for high-performance interprocessor communication has been presented. This protocol supports dynamic, cut-through routing with local flow control, which allows high-throughput, low-latency transmission of packets. In addition, multicast transmissions are allowed, in which a packet is sent to several targets using common resources as much as possible.

The protocol also prescribes mechanisms for detecting and avoiding deadlock conditions due to resource conflicts during multicast. In particular, a copy of the packet is saved before it is split, special packet terminators are used to abort transmissions and trigger retransmissions, and random timeout intervals are used to detect potential deadlock conditions.

Finally, the implementation of this protocol in the CARE simulation system is described. Explicitly representing a packet as the front edge and the terminator allows accurate modelling of word-by-word packet transmission in a functional, event-driven simulator. Also, the success of the implementation indicates that this is a reasonable protocol for interprocessor communication.

References

[1] Gregory T. Byrd, Russell Nakano, and Bruce A. Delagi. *A Point-to-point Multicast Communications Protocol.* Technical Report KSL-87-02, Knowledge Systems Laboratory, Stanford University, January 1987.

[2] V. Ahuja. *Design and Analysis of Computer Communication Networks.* McGraw-Hill, 1982.

[3] P. Kernami and L. Kleinrock. Virtual cut-through: a new computer communication switching technique. *Computer Networks*, 3:267, 1979.

[4] M. Arango, H. Badr, and D. Gelernter. Staged circuit switching. *IEEE Transactions on Computers*, C-34(2):174–180, February 1985.

[5] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[6] P. Kermani and L. Kleinrock. A tradeoff study of switching systems in computer communication networks. *IEEE Transactions on Computers*, C-29:1052, December 1980.

[7] William J. Dally. Wire-efficient VLSI multiprocessor communication networks. In Paul Losleben, editor, *Advanced Research in VLSI—Proceedings of the 1987 Stanford Conference*, pages 391–415, MIT Press, 1987.

[8] Richard W. Watson. Distributed system architecture model. In B. W. Lampson, M. Paul, and H. J. Siegert, editors, *Distributed Systems—Architecture and Implementation*, chapter 2, pages 10–43, Springer-Verlag, 1981.

[9] Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. An *Instrumented Architectural Simulation System*. Technical Report KSL-87-36, Knowledge Systems Laboratory, Stanford University, January 1987.

[10] Sonya Keene and David Moon. Flavors: object-oriented programming on Symbolics computers. In *Common Lisp Conference*, 1985.

Knowledge Systems Laboratory
Report No. KSL-88-10

January 1988

A Performance Comparison of Shared Variables *vs.* Message Passing

Gregory T. Byrd

Department of Electrical Engineering
Stanford University
Stanford, CA 94305

Bruce A. Delagi

Digital Equipment Corporation
Maynard, MA 01754

Submitted for publication to:
ISI Supercomputing Conference
May 1988

A Performance Comparison of Shared Variables *vs.* Message Passing*

Gregory T. Byrd
Stanford University
Stanford, CA 94305

Bruce A. Delagi
Digital Equipment Corporation
Maynard, MA 01754

Abstract

In this paper, we examine the performance of a parallel application implemented in both shared variable and message passing styles. Our purpose is to illuminate the differences between the programming models and show how these differences affect the performance of the programs when executed on systems incorporating hundreds of processing elements.

First, we present the programming models used for the implementations. Then we examine the costs associated with each model, focusing on interprocessor communication and synchronisation. Strategies for minimising data communications costs are discussed and confirmed through simulation. Also, architectural features are identified which have a substantial impact on shared variable and message passing performance.

1 Concurrent Programming Models

Though there is a wide range of concurrent programming models, they can usually be classified according to the *primary* means of communicating between processes. If communication is performed by passing values, we call it a *message passing* model. If communication is done by reading and writing shared memory locations (i.e., passing references), we call it a *shared variable* model.

In this section, we present the details of what we will take as our working example of each of these models. There are certainly other possible models, but they will, for the most part, be mixtures or specializations of the two models presented here.

1.1 The Shared Variable Model

We take *thread-oriented shared variables* to be our primary example of a shared variable model. All communication and synchronization is performed through reading and writing shared variables.

*This material is based on work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was also supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875.

The authors may be contacted at Knowledge Systems Laboratory, 701 Welch Road, Bldg. C, Palo Alto, CA, 94304; or through electronic mail at Byrd@Sumex-Aim.Stanford.EDU or Delagi@Sumex-Aim.Stanford.EDU.

There is one process, or thread of control, for each physical processor involved in the computation, hence the term "thread-oriented."

Various forms of synchronization based on shared variables may be used, including spin locks, semaphores, monitors, barriers, and so forth. In addition, it may be possible for one processor to interrupt another, passing it an interrupt vector which may contain self-referential values (e.g., integers) or references.

When reading or writing global (shared) data, the processor is stalled until a response from the memory system is received.[1] It is presumed that access to global memory is short, compared to the process switch time, so it is more efficient to stall processing than to schedule another process. This means that only one memory request is pending for a given processor at any time.

We do not assume any automatic caching mechanism for shared data, since maintaining cache coherency for large numbers of processors is problematic, and since we want to study alternative compiler- and programmer-directed caching techniques. Instead, globally shared read/write data is declared to be non-cacheable (as in the RP3 parallel computer [10]). Each processor has local (private) memory, and block reads and writes are provided for efficient memory access.

1.2 The Message Passing Model

As representative of message passing styles of computation, we present the *object-oriented streams* model, as embodied in the LAMINA programming language [5]. Objects encapulate local state variables and procedures which manipulate them in response to messages from other objects. Streams represent queues of messages—they are generalizations of *futures* [6], in that a reference to a stream may represent the promise of either a single value or a collection of values to be computed.

The only entities which may be passed through streams are self-referential values (e.g., numbers, symbols, and code bodies), references to streams, and structures, which may have arbitrary internal structure but otherwise contain only self-referential values (as above) or references to streams. Internal structure involving shared substructure is preserved as it is passed between objects.

Synchronization is realized by messages arriving on a stream. Each object has an associated *self-stream*. Whenever a message arrives on an object's self-stream, an execution context for the object is created and control is transferred to the procedure dedicated to handling that message. Execution is, for the most part, taken to completion and is data-driven, although mechanisms for demand-driven computation are also provided.

1.3 References *vs.* Values

In shared variable systems, a reference (or address) is usually given to a location where shared data may be accessed—if the data is needed, it may be read from that location. This is particularly efficient if the needed data is only a piece of a large structure, or if the data associated with the

[1] This restriction maintains the serialisation of memory accesses from a single processor. It is particularly important to guarantee that pending writes have completed: consider the initial conditions a=0, b=0 and the operations aa:=a+1, bb:=b+1, c:="aa greater than or equal bb" if a is the same cell as aa and b is the same cell as bb.[11]

reference need not be accessed at all, but perhaps is passed in turn (as a reference) to another computation which may require the data.

Message passing systems, on the other hand, usually communicate through passing the data itself. If all (or most) of the data is needed for the computation, then this is more efficient, since extra network accesses are not needed. Arbitrary structures, such as graphs, may be passed, but some effort is required on both the sending and receiving ends to linearize the structure for transmission over the network.

Both paradigms, however, recognize the necessity to deal with exceptions to the usual case. In the shared variable model, block transfers may be used for efficient access to vectors of data—less regular structures, however, must still be accessed by "reference chasing." In the stream-based model, data may be encapsulated on a stream, and the reference to the stream passed around until the data is actually needed.

2 Cost Model

In this section, we examine the costs associated with implementations of the two programming models discussed above. Our goal is to identify the costs of program execution in terms of parameters of the underlying multiprocessor system. We then discuss which costs dominate in "efficient" parallel programs, what can be done to minimize the performance degradation due to these costs, and how communication and synchronization overhead relate to overall completion time for a program.

2.1 Evaluation Time

Evaluation time is the term we use to refer to the amount of time spent execting application-level code. If we assume that the same fundamental algorithm is being used in both programming styles, then the amount of application-level work to be done is the same, so the evaluation time should be equivalent.

2.2 Network Communication

We define *network communication* time as the time it takes for a processor to make data available to another processor. In particular, it is *not* the time for one *process* to accept data from another—that will be discussed in the next section.

First, we will introduce a few parameters which characterize the communications network. We assume that the network employs some sort of cut-through routing protocol, such as those described in [7,3,1].

We define the parameter W as the number of cycles it takes to transfer one word over a network channel. Therefore, in the absence of contention, the time to transfer an L-word message[2] over D

[2] L represents the amount of *data* in the message—it does not include the target address.

channels (hops) is $W \cdot D + W \cdot L$, assuming a one-word target address.[3]

If D_{avg} represents the average number of hops traversed by a message and L_{avg} is the average length, in words, of a message, then the average network delay is given as

$$T_{net} = W \cdot D_{avg} + W \cdot L_{avg}.$$

In our thread-oriented shared variable model, accessing globally shared data always causes network activity.[4] As we mentioned earlier, reads and writes cause the processor to block until an acknowledgement is received from the memory module. Every shared variable access, then, requires a round trip through the network.

A one-word read requires sending out a target address ($W \cdot D_{avg}$ cycles) and receiving a two-word (data plus target) response, ($W \cdot D_{avg} + W$ cycles). A write is the same, except that the request takes two words, and the response takes only one. Therefore, a one-word read or write takes $2W \cdot D_{avg} + W$ cycles.

Block reads and writes are similar. A block read request consists of sending an address and a count (2 words) and receiving L words (plus a target address) in return. Though a block write does not require a count, since that is supplied implicitly by the number of values it supplies, we assume that one is supplied explicitly, to handle exceptions and the like. If L_{avg} is the average block size, then the average memory access time is

$$T_{net}^{SV} = 2W \cdot D_{avg} + W \cdot L_{avg} + W.$$

In a message passing environment, however, we are required to pass values with arbitrary internal structure. These values must be encoded into a linear form prior to network transmission. We model this encoding time as a fixed overhead, T_C, plus a constant number of cycles per word, c.[5] For the purposes of this paper, we assume that the encoding operation must be completed before the packet is transmitted (although this is not strictly necessary), so the average network delay for a message is

$$T_{net}^{MP} = W \cdot D_{avg} + W \cdot L_{avg} + (T_C + cL_{avg}).$$

2.3 Process Communication

As mentioned in the previous section, network communication time does not represent the time needed to communicate data between processes. One process must send and the other receive, and there must be coordination between these two phases. (We are not implying synchronous

[3] In a store-and-forward network, the entire packet must be transmitted at each hop, so the latency would be $D \cdot W \cdot (L + 1)$, again with one-word target address.

[4] This is a consequence of not modelling automatic caching of data—shared data must always be read from and written to a memory module. In particular, writes to shared data do not include updating copies of that data in other processors' local memory.

[5] We model encoding/decoding as a linear cost, because we envision an algorithm which uses forwarding pointers [9] to check for shared structures, instead of hashing, which is not necessarily linear, depending on the occupancy of the hash table.

communication, in which the sender/writer waits for the receiver/reader to retrieve the data, but are merely noting that the sending must come before the receiving, and the receiver usually is notified when data becomes available.)

In the object-oriented model, when a message arrives at its target processor, its data is placed on the destination stream, and the object waiting on that stream must be invoked with the proper method. The time required to place a message on a stream is the *queueing* time, T_q. The time required to awaken the necessary object is divided into the *dispatch* time, T_d, which involves selecting the proper object and method, and the *process switch* time, T_{sw}.

Therefore, the total (average) time for communication between objects is

$$\begin{aligned} T_{proc}^{MP} &= T_{net}^{MP} + (T_c + cL_{avg}) + (T_q + T_d + T_{sw}) \\ &= WD_{avg} + W \cdot L_{avg} + 2c \cdot L_{avg} \\ &\quad + 2T_c + T_q + T_d + T_{sw}. \end{aligned}$$

In the shared variable model, communication between two process (A and B) generally takes the following form: A writes a value; A sets a lock; B reads the lock; B reads the data. Thus, communication time depends on what sort of lock is being used. We will consider spin locks at this time, but the analysis may be extended to other synchronization policies.

To estimate the cost of setting and reading a spin lock, consider the optimistic case, where the read is serviced by the memory module just after the write is completed. It takes $WD_{avg} + W$ cycles for the write request to arrive at the memory and $WD_{avg} + W$ cycles to get the result to the reading process, so the overhead time represented by access to a spin lock, T_{lock}^{SV}, is $2WD_{avg} + 2W$.

Thus, the average process communication time using shared variables and spin locks is

$$\begin{aligned} T_{proc}^{SV} &= 2T_{net}^{SV} + T_{lock}^{SV} \\ &= 6WD_{avg} + 2WL_{avg} + 3W. \end{aligned}$$

2.4 Improving Process Communication

One way to lessen the impact of interprocess communication delays is to make efficient use of the network resources—transfer large messages whenever feasible. When L_{avg} is large, in comparison to the other costs, the figures for average latency between processors become approximately $2WL_{avg}$ for shared variables and $(2c + W)L_{avg}$ for message passing.

Thus, when L_{avg} is large, relative to D_{avg}, and when $c = \frac{1}{2}W$, the latency for interprocess communication is the same for shared variable and message passing environments. Notice, however, that the only large "messages" that may be passed in a shared variable system are blocks (vectors) of self-referential values. These structures do not need to be coded, even in a message passing environment, since they are already linear. Thus, if this special case can be recognized by the host machine, coding can be avoided altogether, and the message passing latency becomes approximately WL_{avg}.

Another approach to improving communication would be to decrease D_{avg} and/or W. These are both dependent on the type of interprocessor network used in the system. In high-connectivity

networks, such as hypercubes, the number of hops from one node to another (D_{avg}) is fairly small—$O(\log P)$, for P processors—but pinout and wiring considerations tends to keep the channel width small [2], thus increasing W. Therefore, decreasing D_{avg} by using topology may not decrease the $W \cdot D_{avg}$ product and, in fact, may increase the average latency by increasing $W \cdot L_{avg}$.

Another way to decrease D_{avg} for an application is to exploit locality. If processes (or objects) are placed nearby the data (or other objects) which they need to reference, then the average distance travelled by a message is small. In some cases, good static placement strategies may be developed, based on the network topology and the communication pattern of the application, but in general, determining optimal placement is difficult.

Finally, process communications *throughput* may be improved by overlapping communication with process execution. For example, in the current CARE machine models [4], there are two processors on a processing site—one (the *evaluator*) is concerned with executing application code, while the other (the *operator*) handles communication and process scheduling. Thus message encoding/decoding, network transmission, and process execution may all proceed in parallel. For example, while the operator is encoding a message to be sent to a remote object, the evaluator may be invoking a new object, based on the previous message.

2.5 Completion Time

Using the estimates on interprocess communication developed above, we can estimate a lower bound on the execution time of a parallel program.

2.5.1 Shared Variable

For the shared variable case, the minumum completion time is the sum of the evaluation time, E_{sv}, plus the communication time, $T^{SV}_{proc} \cdot N_{net}$, where N_{net} is the number of interprocess transfers performed by a single process. (One read plus one write counts as a single transfer.)

Thus, a lower bound on completion time is

$$T^{SV}_{comp} \geq E_{sv} + T^{SV}_{proc} \cdot N_{net}.$$

2.5.2 Message Passing

A minimum bound for the object-oriented case is harder to compute, since evaluation, coding, and transmission can all occur in parallel. We consider three cases: (1) when the computation is compute-bound, (2) when coding is the dominant overhead, and (3) when network transmission dominates.

In the first case, the completion time depends merely on the sum of the evaluation and invocation[6] times of all the objects on a site (E_{mp})—communication is completely overlapped with execution. In the second case, completion time is limited by the encoding and decoding of messages. If we assume that there are an equal number of messages sent and received, the bound

[6] See section 2.3.

is twice the coding time $(T_c + c \cdot L_{avg})$ times the number of messages sent by the objects on a site (N_{net}). In the third case, the network is the limiting resource, so the bound becomes the product of the network transit time and the number of messages sent.

Therefore, a minimum bound estimate of the completion time for the message passing model is

$$T_{comp}^{MP} \geq \max \left\{ \begin{array}{l} E_{mp}, \\ 2(T_c + c \cdot L_{avg}) \cdot N_{net}, \\ (W \cdot D_{avg} + W \cdot L_{avg}) \cdot N_{net} \end{array} \right\}.$$

2.6 Summary

The table in figure 1 summarizes the completion time bounds for the shared variable and message passing models, in the general case and when L_{avg} is large. The major differences in the two models are that the shared variable model is much more sensitive to network distance (D_{avg}), while the message passing model is more concerned with message encoding and process switching.

These differences fade away when large blocks of data are being transferred (assuming $c = W$). If we assume that there is a fixed amount of information (dicated by the problem) that must be accessed by the processors for the computation (i.e., the product $L_{avg} \cdot N_{net}$ is a constant), then the most efficient programs will increase L_{avg} and decrease N_{net}. Thus, to a first order of approximation, shared variable and message passing systems deliver the same performance for these efficient programs. To the extent that this type of efficiency is not feasible, performance of the two systems will be largely determined by the factors mentioned above.

3 Experimental Results

In this section, we report the results of a simulation experiment undertaken to explore the performance differences between shared variable and message passing programs. After a brief description of the application, we will present two shared variable implementations and two object-oriented applications. The performance of these programs indicates that, for this application, the costs expressed by the model developed above were in fact the dominating ones for the more efficient implementations developed in both paradigms.

	Shared Variables	*Message Passing*
In general	$E_{sv} + N_{net} \cdot (6WD_{avg} + 2WL_{avg} + 3W)$	$\max \left\{ \begin{array}{l} E_{mp}, \\ 2(T_c + c \cdot L_{avg}) \cdot N_{net}, \\ (W \cdot D_{avg} + W \cdot L_{avg}) \cdot N_{net} \end{array} \right\}$
For large L_{avg}	$E_{sv} + N_{net} \cdot 2WL_{avg}$	$\max \left\{ \begin{array}{l} E_{mp}, \\ 2c \cdot L_{avg} \cdot N_{net}, \\ W \cdot L_{avg} \cdot N_{net} \end{array} \right\}$

Figure 1: Minimum bounds for completion times.

3 *EXPERIMENTAL RESULTS*

3.1 Application Description

The application, called LineSim, is an explicit solution of a system of linear difference equations. The difference equations represent a discretization of the partial differential equations which model the voltage transmission of lossy VLSI metal lines over a substrate.[7]

The wires are divided into segments, where each segment represents an equipotential region and has associated resistance and capacitance parameters. At each time step, a segment's voltages (to the substrate and to the adjacent wire) are computed using its own values and the values of its neighbors calculated during the last time step. The time steps are small enough to guarantee convergence, so there is no need for global synchronization. The segments were divided into rectangular regions, and each region was assigned to a processor. All of the performance numbers presented below are for a 64×64 grid of segments, calculated for ten time steps.

The various implementations of LineSim were all written using the LAMINA programming language [5], which provides parallel extensions to Zetalisp [12] and Flavors [8] for programming in functional, object-oriented, and shared variable paradigms. The programs were executed using the CARE/SIMPLE simulation system [4]. For the experiments described here, the processors were connected in a torus topology—that is, the processing sites were configured in a rectangular grid, with each site connected to its eight neighbors (including diagonal connections), and the edges were wrapped around in the vertical and horizontal directions.

Different machine models were used to execute the shared variable and object-oriented programs. For the object-oriented programs, each processing site corresponds to the model discussed earlier. It contains an evaluator, for executing application code; an operator, for handling communications and process scheduling; a private memory, which is shared by the operator and evaluator; and network components, which actually transmit data across the wires. All of the overhead costs introduced by the message passing cost model are implemented as parameters in the simulation model and are easily varied between (or during) program executions.

In the shared variable model, sites are distinguished as either processing sites or memory sites. The operator on a memory site acts as the memory controller—its accepts requests from the network and sends replies over the network. Some of the overhead costs associated with the operator are ignored, such as packet encoding/decoding, since they are not part of the shared variable cost model. For these experiments, the torus topology was also used for the shared variable programs, with alternate rows representing processing sites and memory sites.

The CARE/SIMPLE system also provides an extensive instrumentation package for monitoring the execution of parallel programs. These instruments display current information about the state of the machine and the program, such as network delays, execution times, and so forth. Most of the numbers discussed below were obtained by the simulator's instrumentation.

3.2 Shared Variable Implementations

Two shared variable implemetations of LineSim were developed, called `sv-point` and `sv-block`. Their execution times for systems ranging from four processors to 256 are shown in figure 2.

[7] This is essentially the parabolic PDE system represented by the "diffusion" equation.

3 EXPERIMENTAL RESULTS

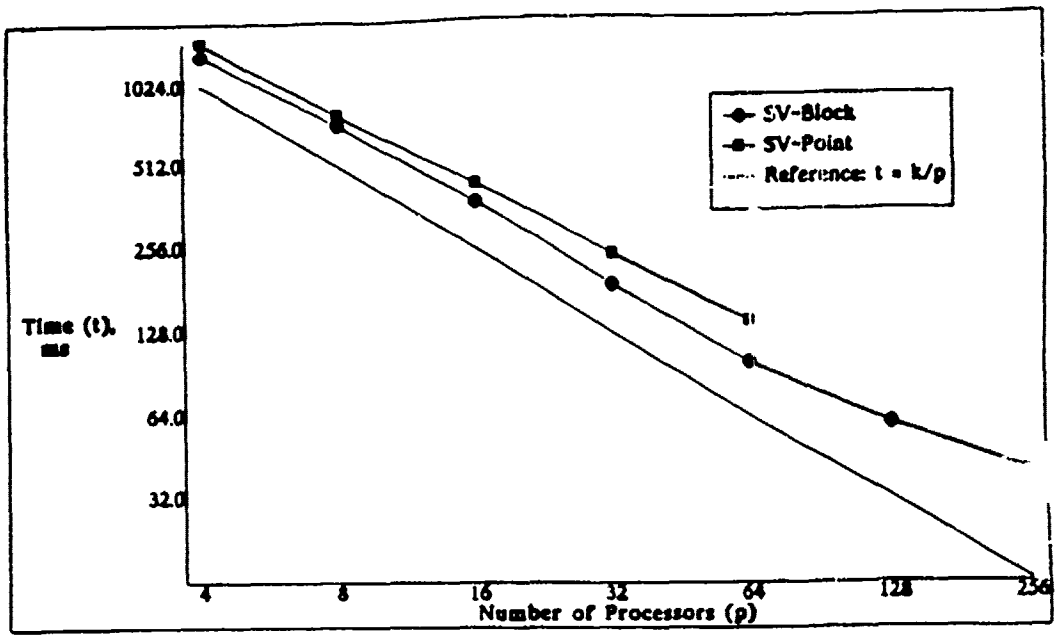


Figure 2: Shared variable LineSim completion times.

Both implementations use the thread model—one process per processor. Each process has a local array containing the voltages for the segments in its block. Only the values for segments along the edges of the block need to be shared.

The basic operation of the threads is as follows:

1. Get the edge values of the neighboring blocks, and store them in the local voltage array.
2. Calculate the segment voltages for this time step.
3. Write the edge values of this block to global memory.

Both implementations use spin locks to synchronize with its neighbors. When a thread is ready to read a neighbor's edge values, it reads a location associated with that thread until its value corresponds with the current time step. Similarly, when the thread writes its own edge values, it increments the lock locations for its neighbors.

The difference in the two implementations is strictly in how its values are read and written. The `sv-point` thread reads in voltages one at a time, while `sv-block` uses block transfers. Since block transfers allow fewer network accesses, they incur less overhead cost than word transfers. In view of the cost model, terms involving D_{avg} are dominating in `sv-point`, while terms involving L_{avg} dominate in `sv-block`.

The relative performance of the two programs becomes more disparate as the number of processors increases, because the computation time decreases by a factor of 2 with each doubling of processors, while the communication time only decreases 25%. This is because the computation

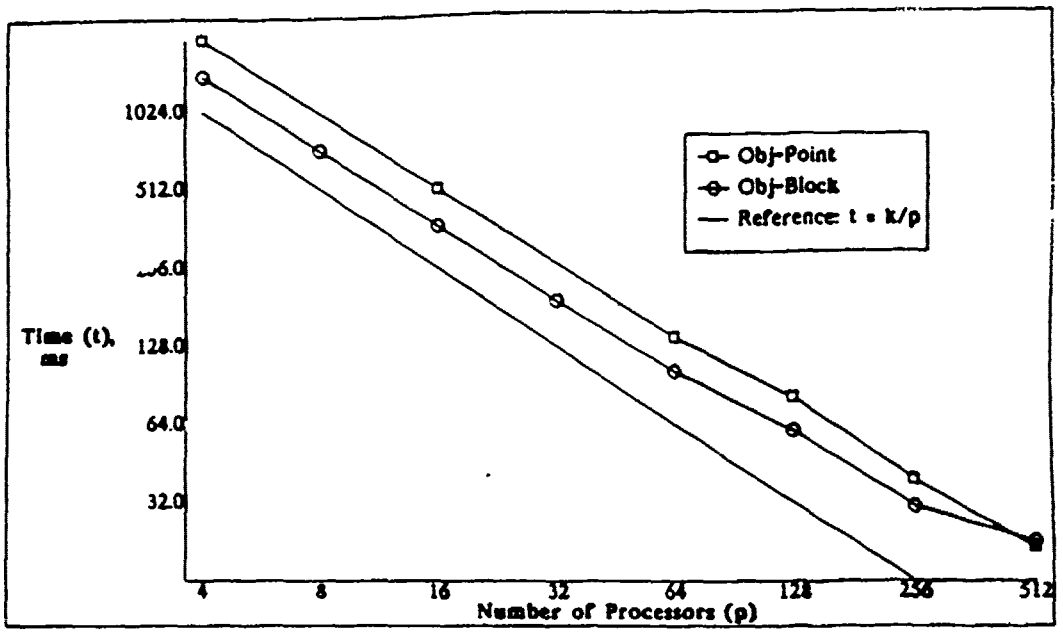


Figure 3: Object-oriented LineSim completion times.

depends on the area of the region of segments, while communication depends on its perimeter. (No performance data was taken for `sv-point` at 256 processors, because of excessive simulation time required.)

One final observation about these programs has to do with the effectiveness of automatic caching. In this application, the naive thing to do would be to store all the segment voltages in a global array. Assuming the cache is big enough, all of the values for the local segments would have been collocated with the processor after the first iteration. However, some edges are represented by rows of the array and some by columns. If the array were stored in a row-major fashion, for example, a block transfer of a column would not be accomplished just by fetching the first value—instead, a lot of interior points would be needlessly transferred to the other cache, and would have to be transferred back when they had to be written. Unless the shared variables were recognized as shared and set apart as vectors, which would be fetched as blocks by a caching system, there would be much wasted network bandwidth.

3.3 Object-Oriented Implementations

Two implementations were developed in the object-oriented style. These are `obj-point` and `obj-block`, and their performance numbers are shown in figure 3.

`Obj-point` represents the naive object-oriented solution to this problem, in which each segment is represented as a separate object. As before, the segment waits for values from its neighbors, calculates its new voltages, and then sends the updated values to its neighbors. This time, however, the computation and communication are on a per segment basis.

There are two potential problems with this approach. First, each segment uses message passing to send new values to all of its neighbors, even when they reside on the same processor site. In fact, at least two neighbors are *guaranteed* to be on the same site. This needlessly consumes operator resources. Second, the process switching cost is incurred for every segment computation, rather than being amortized over all the computations for a block.

As it turns out, for the nominal values of the machine parameters chosen, the second problem is the critical one—since the operator handles packets in parallel with execution, and since the evaluator has lots of objects to deal with, the communication system usually manages to keep up. However, the process switching overhead represents a fair percentage of the computation time, so speedup is degraded.

`obj-block`, on the other hand, represents a *block* of segments as an object. As in the shared variable implementation, only the edge values are communicated. When all the edges are received, the segment calculations are performed, and the updated edge values are sent to the neighboring blocks.

The block-oriented implementation consistently out-performs the segment-oriented version over a wide range of system size. When the number of processors gets very large, however, the number of segments per processor gets small, and `obj-point` becomes more efficient. For this problem, the crossover occurs between 256 processors (16 segments/processor) and 512 processors (8 segments/processor).

3.4 A Closer Look

Figure 4 shows the performance of all four LineSim implementations. (Again, the shared variable programs were not run for large numbers of processors because of excessive simulation time.) The two best implementations, `sv-block` and `obj-block`, show essentially the same performance through 128 processors.

Since the process granularity is so large, small differences in overhead would tend to be swamped out. Therefore, we measured the performance using an evaluator which is 100 times faster than the earlier runs. This greatly decreases the computation time, while communication time stays fixed, so differences in communication overhead should be more evident.

Figure 5 shows the completion times of the two block-oriented programs for 64 processors, with increased processing speed. First, each program was run with the nominal values used by our simulation system, namely $W = 16$ (corresponding to 4-bit channels) and $c = 16$. (The value of c is ignored in the shared variable runs.) For this program, $D_{avg} = 1$, and $L_{avg} = 32$.

Using the default values, the shared variable program did slightly worse, due to the fact that execution may not proceed in parallel with communication. The dominating costs in the message passing program, according to the cost model, is coding time.

Next, the value of W was decreased to two cycles per word. The model would suggest that decreasing W would have a greater effect on the shared variable model (due to more trips through the network) than on the message passing model. As expected, the shared variable completion time decreased by 70%, while the object-oriented program showed only an 11% improvement.

3 EXPERIMENTAL RESULTS

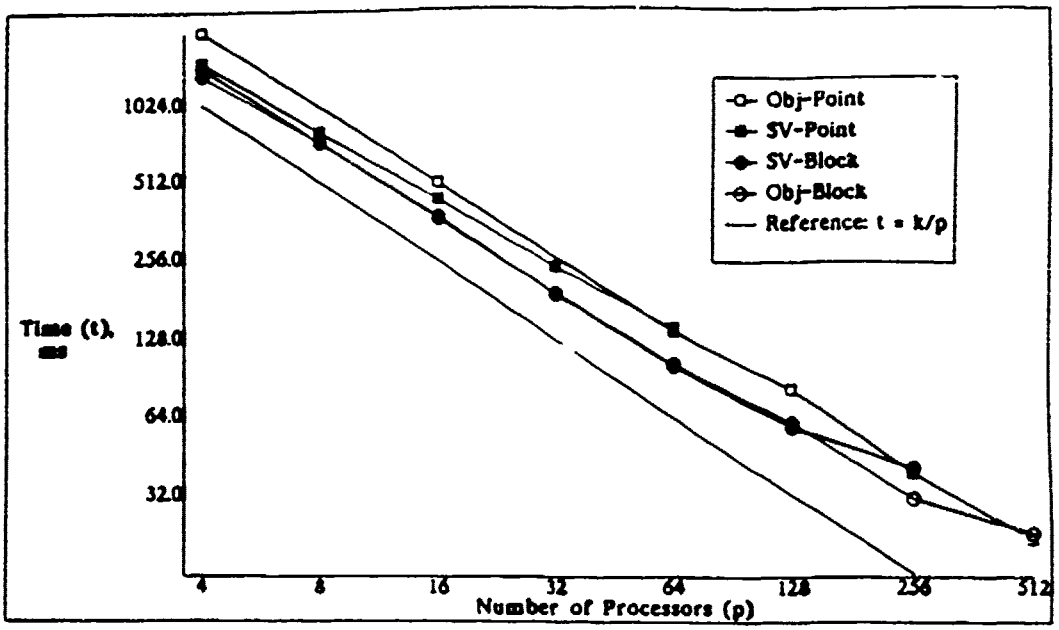


Figure 4: LineSim completion times.

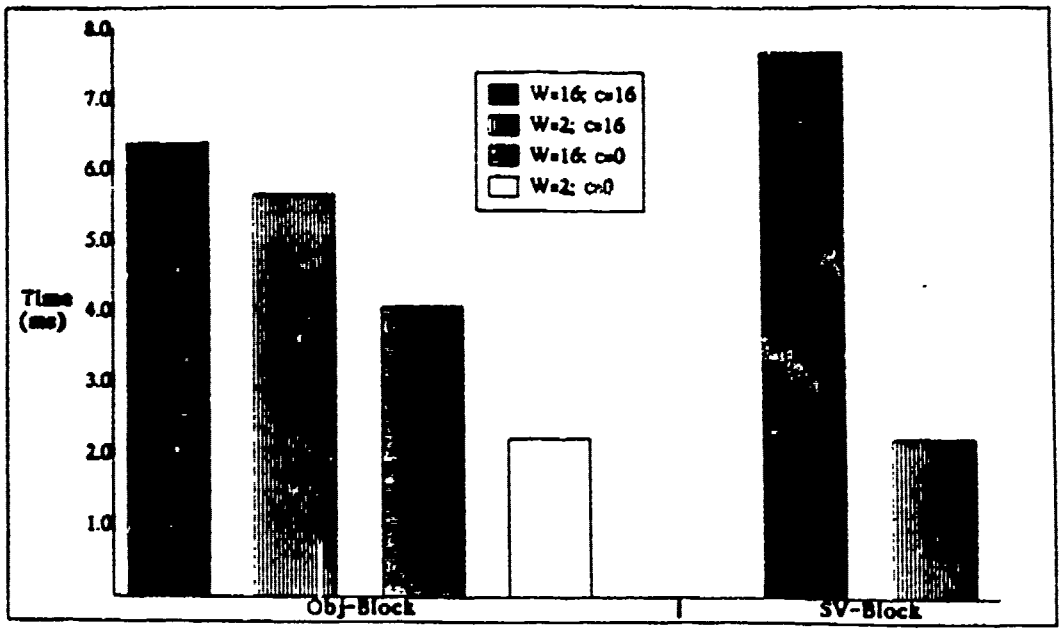


Figure 5: Completion times with 100x evaluator speedup.

However, decreasing coding time to zero (by not coding vectors, for example) had a more pronounced affect on the message passing program. The dominant costs moved to the network, resulting in a performance increase of 36%. When the zero coding time is combined with large channel width, the two programs have virtually identical performance.

4 Conclusions

The goal of this experiment was to understand some of the potential performance differences for message passing and shared variable programs. In particular, we looked at the implications of the communications mechanisms in both paradigms.

The analytic cost model developed in this paper appears to provide a reasonably good first approximation to the costs of data communication in parallel programs. Many factors, however, are not included in the model. Some of these factors, such as network contention and routing strategy, are modelled by the simulation system. The close agreement between the analytic model and the results supplied by the simulator demonstrate that, for this application, those costs did not dominate the performance. System factors, such as paging costs and global resource allocation and reclamation, were not included in the models or in the simulations. The costs associated with the system considerations were assumed not to dominate performance.

The cost model developed here is an attempt to quantify the difference in overhead costs for the two programming models. The shared variable model is particularly sensitive to network latencies, while packet formatting is a greater concern in a message passing system. The experimental results are not meant to be conclusive evidence of the superiority of one programming style over another, but they do offer an indication of the important machine parameters to be optimized to support one or both of these paradigms.

Acknowledgments

The authors would like to thank Nakul Saraiya for his extensive work on the simulation system, and in particular the shared variable interface, and Sayuri Nishimura, whose efforts in developing and supporting the instrumentation system have allowed meaningful measurements to be taken. We also acknowledge the many users and developers of SIMPLE/CARE for helping to enrich the system and direct its growth.

References

[1] Gregory T. Byrd, Russell Nakano, and Bruce A. Delagi. *A Point-to-point Multicast Communications Protocol.* Technical Report KSL-87-02, Knowledge Systems Laboratory, Stanford University, January 1987.

REFERENCES

[2] William J. Dally. Wire-efficient VLSI multiprocessor communication networks. In Paul Losleben, editor, *Advanced Research in VLSI—Proceedings of the 1987 Stanford Conference*, pages 391–415, MIT Press, 1987.

[3] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[4] Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. *An Instrumented Architectural Simulation System.* Technical Report KSL-86-36, Knowledge Systems Laboratory, Stanford University, January 1987.

[5] Bruce A. Delagi, Nakul P. Saraiya, and Gregory T. Byrd. *LAMINA: CARE Applications Interface.* Technical Report KSL-86-67, Knowledge Systems Laboratory, Computer Science Department, Stanford University, November 1987.

[6] Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 1984.

[7] P. Kernami and L. Kleinrock. Virtual cut-through: a new computer communication switching technique. *Computer Networks*, 3:267, 1979.

[8] David A. Moon. Object oriented programming with Flavors. In *Object-Oriented Programming Systems, Languages, and Applications [OOPSLA] '86 Proceedings*, pages 1–8, September 1986.

[9] David A. Moon. Symbolics architecture. *Computer*, 20(1):43–52, January 1987.

[10] G. F. Pfister, *et. al.* The IBM Research Parallel Processor Prototype (RP3): introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, IEEE, 1985.

[11] Marc Snir. October 1986. In lecture, Stanford University.

[12] Daniel Weinreb and David Moon. *Lisp Machine Manual.* Symbolics, Inc., Cambridge, MA, 1981.

Knowledge Systems Laboratory
Report No. KSL-88-81 (revised)

June 1989

Multicast Communication in Multiprocessor Systems

Gregory T. Byrd and Nakul P. Saraiya

Stanford University
Stanford, CA 94305

Bruce A. Delagi

Digital Equipment Corporation
Palo Alto, CA 94301

To appear in:
1989 International Conference on Parallel Processing

Multicast Communication in Multiprocessor Systems *

Gregory T. Byrd and Nakul P. Saraiya
Knowledge Systems Laboratory
Stanford University, Stanford, CA 94305

Bruce A. Delagi
Digital Equipment Corporation
Palo Alto, CA 94301

Abstract: Recent high-performance multiprocessors exploit cut-through routing for unicast transmission, with packets routed as their first bytes arrive. We extend ideas considered for efficient cut-through routing in multiprocessor systems to include multicast, in order to benefit the many parallel programs in which producers provide each value to multiple consumers. We describe several alternative cut-through multicast protocols, including a restrictive (yet adaptive) routing scheme for deadlock avoidance. Simulations using synthetic and application-driven loads show it has significantly better performance than either multicast emulation or deadlock detection and resolution. The scheme provides cut-through multicast without requiring dedicated storage in the communication facilities for a full packet.

1 Introduction

The communication patterns naturally found in parallel programs include those in which a producer provides values to more than one consumer [4]. These patterns can be directly supported by communication facility routing protocols or indirectly supported by arranging that either the operating system or the application build a tree of communicating processes whose leaves are the "real" targets of communication. Intermediate nodes of this tree store packets and forward them to the next level of the tree. Such store and forward techniques may take limited advantage of available network facilities in a system providing cut-through routing [6]. Additionally, interrupt handling (and possibly process switching) latencies are incurred at each forwarding step. These can be significantly larger than the transmission time of the packet itself. These performance considerations motivated us to study means to provide direct support for multicast communication.

*This work was supported by Digital Equipment Corporation, by DARPA Contract F30602-85-C-0012, by NASA Ames Contract NCC 2-220-S1, and by Boeing Contract W266875. G. Byrd was supported by a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

1.1 Unicast Protocol

The unicast communication protocol underlying the multicast facilities discussed here has been described in [1]. It includes provision for adaptive routing and is deadlock-free so long as the *computing nodes* (see figure 1) of the system have sufficient available storage to hold the blocked packets. The communication facilities themselves provide only enough buffering at each *port* to hold a packet target address. Flow control is done in units of transmission activity equal to one of these buffers. Independent routing decisions based on local path availability information are made by each *router* encountered by a packet as its front edge makes progress from its source to its target.

For normal operation, only small amounts of dedicated resources are used in the transmission and

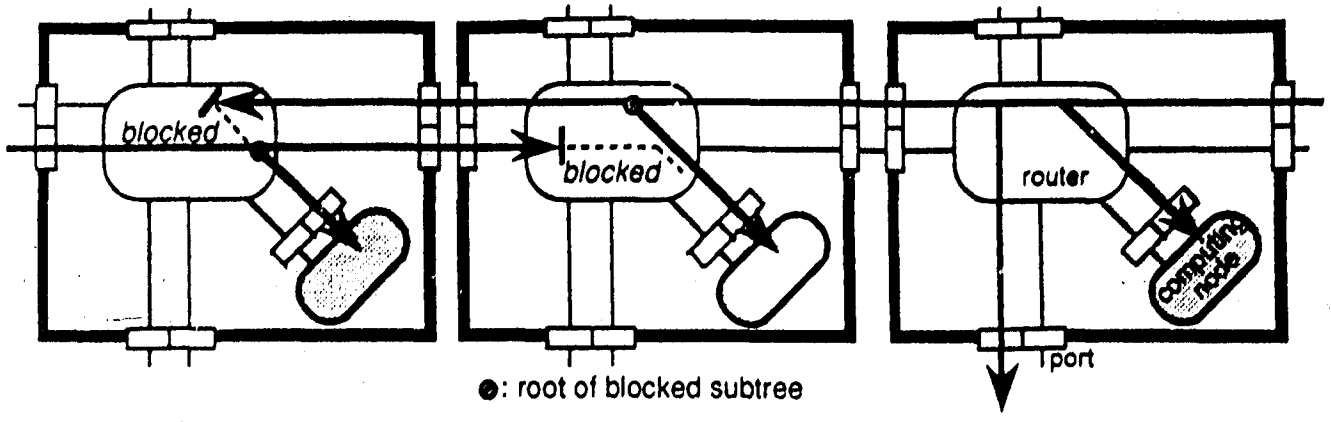


Figure 1: Multicast Deadlock

reception of information, namely the small amount of buffering provided at each port. In the exceptional condition that no suitable output port is available for an incoming packet, additional storage is made available by contending for it from a buffer pool managed by the local computing node.[1] The packet is then stored for forwarding to its target at a later time. The use of small dedicated buffers for normal operation together with larger buffers allocated from a common pool of storage to handle exceptional conditions permits high performance with simple, low cost, implementations.

1.2 Multicast Deadlock

If direct support of multiple consumer communication patterns in concurrent programs is to be provided, the possibility of deadlock must be considered. As shown in figure 1, when two multicasts have each acquired some paths also needed by the other, each will block progress by the other: a deadlock is the result.

One way to handle deadlock is to ensure that no path may be blocked; that is to associate with each input port enough dedicated storage to buffer the largest possible message. Motivated as discussed above, however, our design goal was for the multicast communication facility to have little dedicated storage relative to anticipated packet sizes.[2]

We will consider three alternative approaches for dealing with deadlock: (1) emulating multicast by multiple unicasts—eliminating the problem by effectively eliminating the facility, (2) detecting and resolving deadlocks, or (3) avoiding deadlock. We call the first approach *multi-unicast* (MU). For the second approach, the deadlock detection and resolution scheme we consider allows all but the potentially deadlocked subtree to proceed. The transmission of just this subtree is aborted and later resumed. (Two such subtrees are shown in figure 1.) We call this scheme *resumable multicast* (RM). Finally, the third approach—*restricted branch multicast* (RBM)—provides deadlock avoidance by restricting routing alternatives.

1.3 What Will Be Shown?

In the remainder of this paper we will describe the details of each of the three schemes and present their simulated performance. Our studies, using

[1] Through the use of virtual channels [2] we could eliminate the requirement for such buffers at the cost of consuming network resources holding delayed packets in the net and with significant impact on the possibilities for adaptive routing.

[2] As indicated, the dedicated storage in the transmission path need only be large enough to hold a target address. Breaking packets into sub-packets introduces significant partition and re-assembly overhead and entails substantial proportionate increase in such dedicated storage.

the CARE simulation system [3], include both test cases contrived to exercise multicast facilities and also an application characterized by a mix of multicast and unicast communication. Based on our expectations of the relative times required to drive signals within and between sites in multiprocessor systems, the computing nodes in these simulations execute instructions four times faster than the network ports accept, transmit, and deliver input. Based on expected die pin count limitations, four pairs of 16-bit network ports are associated with each site. The computing node at each site is assumed to include a resource for message handling (including forwarding) that is independent of the resource doing computation.

We will discuss the performance of the communications system in each of the three schemes in terms of the measured *network latency* for the packets it handles. Network latency is defined in this paper as the difference between the time a packet is made available for transmission and the time it is completely received at its target computing node. If, due to unavailability of a suitable output port, a packet is temporarily stored at a computing node which is not its target, the time taken to forward the packet (as well as the time spent waiting to be forwarded) is included in the network latency for that packet. By this measure, the restricted branch approach to multicast transmission (RBM) will be shown in many cases to have significantly better performance than the other two schemes considered.

2 Multicast Protocols

In this section, we describe the three multicast protocols and give expressions for the average network latency in the absence of contention. Then, in order to assess the performance of each approach in the presence of network load, we compare the latencies experienced while running a test program designed to exercise the multicast facilities.

The latency of a *unicast* packet in a cut-through network, in the absence of contention, is $T = \lceil \frac{t}{W} \rceil D + \lceil \frac{L}{W} \rceil$, where: t is the number of bits in a *target*, i.e., a network address; D is the *distance* (the number of channels) traversed by the packet; L is the *length* of the packet (data only), in bits; and W is the *width* of the network channels, in bits.

The above equation gives latency in terms of *cycles*, where a cycle is the time to transfer W bits of data from an input port of one site to an input port on the next site. In describing the latencies experienced by the multicast schemes, we will use the symbols $\bar{D}$ and $\bar{L}$ to denote the *average* distance and length, respectively.

2.1 Multi-Unicast

The simplest form of support for multicast, the *multi-unicast* (MU) protocol, emulates it by transmitting the entire packet sequentially to each target.

In an MU transmission with n targets, the average packet must wait for the $(n-1)/2$ packets ahead of it to be delivered to the network. Adding this to the network delay experienced by the average packet yields

$$T_{\mathrm{MU}} = \left\lceil \frac{t}{W} \right\rceil D + \left\lceil \frac{\bar{L}}{W} \right\rceil + \frac{n-1}{2} \left\lceil \frac{t}{W} \right\rceil + \frac{n-1}{2} \left\lceil \frac{\bar{L}}{W} \right\rceil .$$

For large n or $\bar{L}$, the cost of placing multiple copies of the packet onto the network is likely to dominate the average latency.

2.2 Resumable Multicast

Multi-unicast creates n independent copies of a packet to be sent to n destinations. If the transmission paths between the source and several destinations have some set of channels in common, MU uses those channels inefficiently by transmitting the same data multiple times. A more efficient approach might be to transmit along a common path as much as possible, then "split" the packet by transmitting the data down several paths simultaneously.

This is the approach taken by the *resumable multicast* (RM) scheme. Because of the limited buffering at network ports, data can only be transmitted if all paths are able to receive it. If one path is blocked, it must become unblocked before any of

its siblings may proceed. As described earlier, this can lead to deadlock (see figure 1)—neither multicast can proceed, since each has acquired resources needed by the other. In RM, each site at which a split has occurred can detect a potential deadlock situation (e.g., if the number of consecutive blocked cycles has exceeded some threshold) and request that all the downstream transmissions from that point be aborted.

To recover from the abort, a local copy of the packet is always kept and may be retransmitted at a later time. Since the network ports themselves do not have sufficient storage to temporarily buffer the packet, space must be allocated from the buffer pool of the site's computing node. If space is not available, or if access to the computing node is blocked, the packet is not split but is merely "passed through" toward the first target in the packet's target list.

In the absence of contention, the average RM latency is very close to the unicast latency. However, an RM packet contains $n-1$ additional targets, relative to a unicast packet. Thus, the average latency equation is

$$T_{\mathrm{RM}} = \left\lceil \frac{t}{W} \right\rceil \bar{D} + \left\lceil \frac{\bar{L}}{W} \right\rceil + (n-1) \left\lceil \frac{t}{W} \right\rceil .$$

The targets are now transmitted in series with the data, rather than sending a copy of the data with each target. The number of targets, then, is an *additive* component in the latency, rather than *multiplicative*, as we found in the MU case.

2.3 Restricted Branch Multicast

Resumable multicasts can be very sensitive to network load, for three reasons. First, they split aggressively, thus increasing the chances that one branch will be blocked due to contention with some other packet in the network. Second, each split consumes the port connecting to the computing node at a site, regardless of whether there are any local targets, this precludes any other *useful* packet (multicast or unicast) from being accepted at that site. Third, *all branches* of a blocked subtree are aborted and retransmitted, thus wasting the resources they have seized as well as excluding other messages from using them. Aborts thus add to the network congestion, which can cause more aborts—positive feedback which can degrade the performance of the whole network.

This lead us to consider a third multicast protocol, the *restricted branch multicast* (RBM). The main characteristics of this protocol are (1) a reduction in the fanout of a split packet, and (2) a scheme for deadlock *avoidance*, rather than deadlock detection and resolution.

In RBM, a packet may split into at most two paths at any site. In our simulated implementation, one of these paths must be *local*—that is, it includes the port connected to the computing node at that site. The single non-local path is determined by the first non-local target in the packet, and *all* non-local targets are routed along that path.

This restriction in itself does not prevent us from getting into deadlock—in figure 1, each split satisfies the conditions described above, yet deadlock occurs. To avoid deadlock, we provide an additional port, called the *split port*, to the computing node at each site, to be used exclusively for the local path of a split packet. If this port is not available, the packet may not be split. It may be buffered completely at the local site (using the normal *unicast port* to access the computing node), in which case the local targets will be stripped off and the packet retransmitted later, or it may pass through, without servicing any local targets.

Since it cannot split, a packet which represents the leaf of a multicast is identical to a unicast packet and therefore must utilize the unicast port (not the split port) to connect to the computing node. If busy, the unicast port will eventually become free, since no multicast packet can use it, and all packets are of finite size. Therefore, no multicast will block indefinitely, and deadlock is avoided. (Two local ports are sufficient, because we only allow two-way splits. Higher fanout would require additional ports.)

To calculate the average latency for RBM, we first assume that the distance between any pair of targets is $\bar{D}$. Second, we assume that the chances of encountering another target on the path is small—i.e., a packet is *only* split when it reaches the first

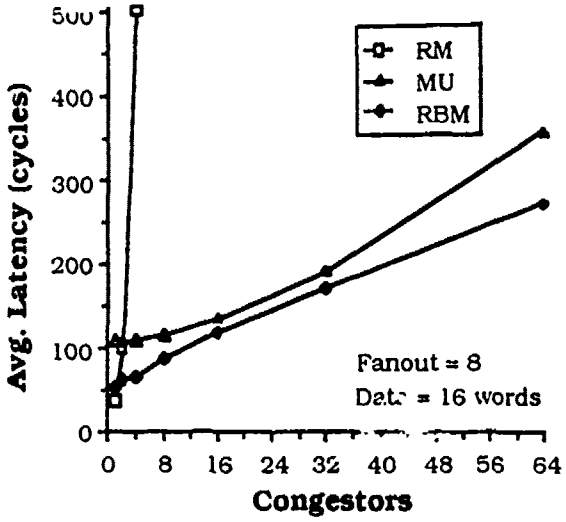


Figure 2: `congest`: Variable congestors.

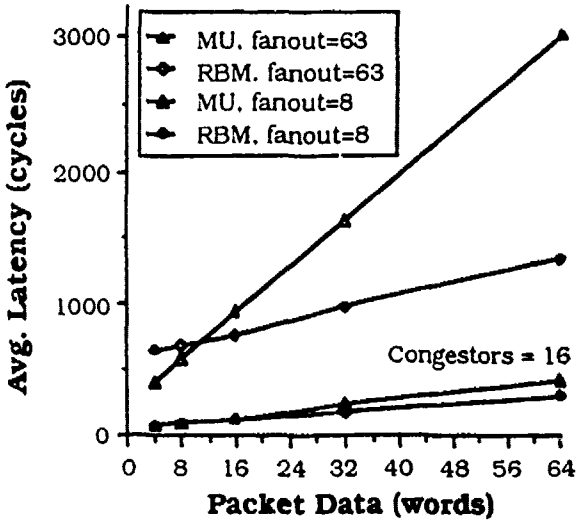


Figure 3: `congest`: Variable packet size.

target on its current target list. The head of the average RBM packet, then, will travel $\frac{1}{2}(n+1)\bar{D}$ hops to arrive at its target site. As in the RM case, the packet length must take into account the additional targets. For the average packet, half the targets will have been split off when its destination is reached, giving a packet length of $\bar{L} + \frac{1}{2}nt$. Thus, the latency of the average packet in an RBM multicast is given by

$$T_{\mathrm{RBM}} = \frac{n+1}{2}\left\lceil \frac{t}{W} \right\rceil \bar{D} + \left\lceil \frac{\bar{L}}{W} \right\rceil + \frac{n-1}{2}\left\lceil \frac{t}{W} \right\rceil .$$

The only difference between this and the multiunicast latency (T_{MU}) is an $O(nt\bar{D})$ term in place of an $O(n\bar{L})$ term. Thus, if $\bar{L} > t\bar{D}$, RBM performs better than MU for a single multicast to randomly distributed targets. In the best case, all targets lie on the path between the source and the farthest target, in which case the total number of channels traversed equals the distance to the farthest target, and the average latency is the same as in the resumable multicast case.

3 Benchmark Performance

According to the latency equations developed above, one would expect resumable multicast to be the clear winner. If we make the reasonable assumptions that $\bar{L} \geq t$ and $\bar{D} \geq 1$, it will never have a higher latency than the other schemes. Those equations, however, assumed no contention in the network. To study how the three multicast schemes performed under load, we ran a test program called `congest`.

The `congest` program creates a number of processes, called *congestors*, each of which simultaneously multicasts a packet to a fixed number of other sites. While holding the network size and topology constant, there are three ways of increasing network load using `congest`: increasing the number of congestors, increasing the multicast fanout, and increasing the packet size.

We simulated the executing of `congest` on an 8×8 torus. The congestors were placed randomly, one per site; the same random placement was used for all experiments. Finally, the multicast targets were picked randomly, and the order of the target list was also random. Figure 2 shows the average

latency for each scheme as the number of congestors was increased from one to 64, with a fanout of eight. For a fanout of 63 (i.e., broadcast), the relative performance of the three protocols was almost identical to the results shown. Figure 3 shows the effect of varying the packet size (data only) from one to 64 words, with sixteen congestors and fanouts of eight and 63.

Resumable multicast predictably shows the lowest latencies for a single multicast, but its performance degrades dramatically as the number of congestors is increased. This is not surprising, especially in the high fanout case, since simultaneous multicasts are very likely to interfere with one another, resulting in a high number of aborted packets.

The average latency for *multi-unicast* is almost constant for one, two, and four congestors—since there are few packets in the network at one time, there is little degradation due to network contention. The dominant cost in these cases is the time to place the multiple packets on the network. At sixteen congestors and above, the effects of network load become significant. Also, as predicted by the latency equations above, the average latency for MU is a strong function of packet size, especially for large fanout.

Restricted branch multicast shows the lowest average latency in almost every multi-congestor configuration. It is competitive with RM for the single-multicast case, and clearly exhibits better performance under load. With respect to MU, the effects of network contention are more pronounced for RBM under light loads. Each time an RBM packet must be temporarily stored and forwarded, the retransmission delay may effect more than one target (compared to only one target in MU). Thus, for **congest**, retransmission has a greater effect on the average latency. However, RBM still wins, except for very small packets, since it does not have the overhead cost of handling multiple packets at the source.

With small packets under high loads, MU shows lower average latencies than RBM. This is partly due to the impact of retransmission costs, discussed above. Additionally, the $O(n\bar{L})$ overhead of multiple packets is not as great when $\bar{L}$ is small. With smaller fanout (and thus lower load), the performance of MU and RBM are almost identical for small packets.

A final distinction between MU and RBM is the rate at which the source can deliver messages to the network. In the MU scheme, the source must place n packets sequentially onto the network, so that no other packet may be sent for $O(n(\bar{L}+t))$ cycles. With RBM, the next message may be sent after $O(\bar{L}+nt)$ cycles. The rate at which messages may be sent to the network may limit the effective granularity of the computation at a site.

While it is interesting to study the behavior of these multicast protocols under varying loads, we recognize that **congest** is a contrived test case and may be pessimal, especially towards RM. In the next section, we will examine how the protocols perform in the context of a more realistic application program.

4 Application Performance

We studied the performance of the three multicast schemes in the context of ELINT[4], a real-time system for interpreting radar emissions from aircraft. The application correlates streams of radar emissions that have been observed by detection sites into radar emitters. These are grouped into clusters that are tracking together, and the activity of inferred aircraft is monitored. ELINT is implemented in a concurrent, object-oriented language called LAMINA[5], in which objects respond to messages by executing data-driven, run-to-completion tasks.

ELINT uses pipelines of dynamically created objects to represent the hypothesized emitters and clusters in the airspace. Objects are replicated as necessary to widen pipelines and relieve congestion. This introduces the need to send the same data to multiple places, for example, in propagating timestamps to emitter pipelines, or in matching an emitter against a cluster, which is represented by replicated objects. ELINT relies on a multicast facility to provide this service.

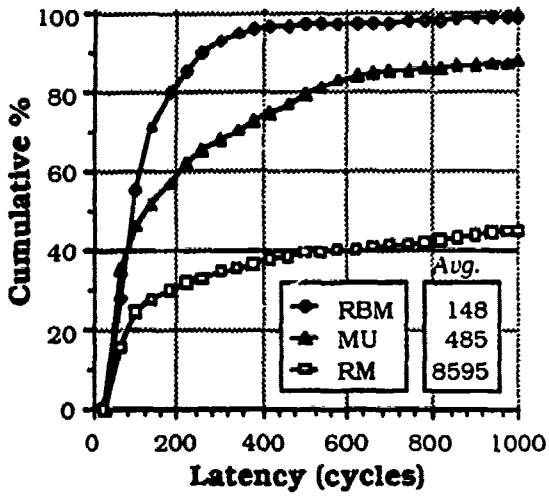


Figure 4: ELINT: Multicast latency distributions.

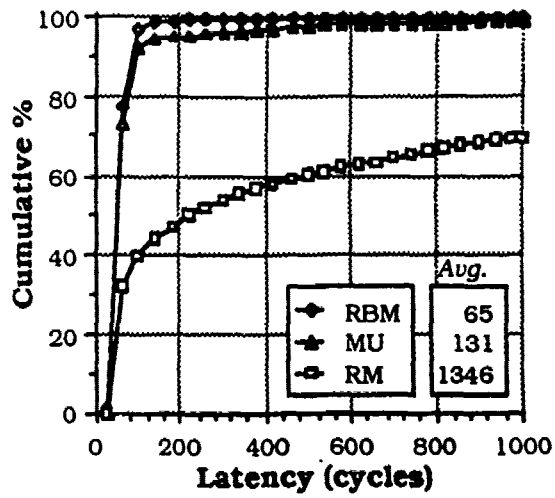


Figure 5: ELINT: Unicast latency distributions.

4.1 Results

For the experiment reported here, using a 16 x 16 torus, a typical object processed a 25–35 word message in 1500–2500 processor cycles before sending approximately one message of about the same size. The sources and sinks of messages were randomly distributed because objects were randomly sited as they were created. Multicasts constituted 7–9 percent of all transmissions and 25–30 percent of all receptions, with an average fanout of 4 and a maximum of 30 targets. The network latency distributions of multicast and unicast packets are shown in figures 4 and 5, respectively.

Multicasts performed best with the restricted branch scheme, achieving an average latency that was one-third that of multi-unicasts. Resumable multicasts performed very poorly, showing an average latency that was more than an order of magnitude higher than the others.

The majority of unicasts displayed virtually identical performance with both multi-unicast and restricted branch multicast, although the average latency was half as much with restricted branch because of the smaller tail in the distribution. Resumable multicasts, on the other hand, significantly impacted unicasts and increased their average latency by an order of magnitude. To understand these results, let us consider each multicast approach in turn.

Multi-Unicast. The $O(n\bar{L})$ source delay involved in injecting multicasts into the network is the major drawback of MU. Besides affecting the constituent messages of the multicast as discussed earlier, it also impacts messages that are queued behind it, be they messages initiated by the computing node at that site or those that are temporarily stored at the site due to network congestion. In ELINT, $\bar{L}$ was sufficiently large and multicasts sufficiently frequent for this to be visible in the latency of both multicasts and unicasts.

Resumable Multicast. As discussed earlier, resumable multicasts both increase network load and perform poorly as a result of such load. This was particularly evident in ELINT, for two reasons. First, some high-fanout multicasts were driven by the arrival of input data (*e.g.*, the distribution of timestamps to emitters); the constant data arrival rate led to the generation of multicasts at a constant rate that was higher than the rate at which they could be delivered. Second, multicasts often occurred in bursts. The net result was the same in

both cases—increasingly severe contention for limited resources, to the detriment of all. In fact, the only reason many multicasts were successfully delivered with RM was that the program accepted only a finite amount of data so that new multicasts stopped being generated, allowing those in transit to finally complete. Hence, the "average" latency of RM multicasts reported here may be unduly optimistic.

Restricted Branch Multicast. There are two potential drawbacks with the RBM approach. The first is the $O(n\bar{D})$ latency that is incurred as the front edge of a multicast visits each target in turn. For the 16 x 16 torus, $\bar{D}$ is approximately 8; this was sufficiently small relative to $\bar{L}$ to make this latency tolerable. The second problem is that the delay incurred by a multicast packet that is stored and forwarded due to network congestion is also incurred by all its constituent messages (targets). In ELINT, 5–7 percent of all packets (multicasts and unicasts) were stored and forwarded for both RBM and MU. This affected twice as many messages using the RBM scheme, but the average added delay per packet was higher for multi-unicast, due to the source delay discussed above.

5 Conclusions and Future Work

The experiments reported here represent only the first step toward a complete evaluation of the costs and benefits of network-supported multicast. More work is needed to understand the performance of these protocols for applications and systems with different network load characteristics, such as the multicast invalidate traffic of a directory-based cache coherence scheme. Additionally, a detailed analysis of the hardware cost is required. Finally, performance enhancements should be considered, such as sorting the targets of an RBM packet to specify an optimal path or adding resources to reduce the cost of retransmitting temporarily buffered packets.

Nevertheless, the data presented here suggests that multicast transmission can be directly supported in multicomputer systems which utilize communication networks based on cut-through routing with minimal buffering. The restricted branch multicast protocol provides lower latency than a multi-unicast approach, over a significant range of network conditions. While the resumable multicast protocol provides good performance in isolation, its behavior in the presence of network contention appears to make it unsuitable for general use.

References

[1] G. Byrd, R. Nakano, and B. Delagi. A dynamic cut-through communication protocol with multicast. Technical Report STAN-CS-87-1178, Dept. of Computer Science, Stanford Univ., Aug. 1987.

[2] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.*, C-36(5):547–553, May 1987.

[3] B. Delagi et al. An instrumented architectural simulation system. Technical Report STAN-CS-87-1148, Dept. of Computer Science, Stanford Univ., Jan. 1987.

[4] B. Delagi and N. Saraiya. ELINT in LAMINA: Application of a concurrent object language. *SIGPLAN Notices*, Apr. 1989.

[5] B. Delagi, N Saraiya, and G. Byrd. LAMINA: CARE applications interface. In *3rd Intl. Supercomputing Conference*. Intl. Supercomputing Institute, May 1988

[6] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

Knowledge Systems Laboratory
Report No. KSL-89-15

March 1989

Support for Fine-Grained Message Passing in Shared Memory Multiprocessors

Gregory T. Byrd

Department of Electrical Engineering
Stanford University

Bruce A. Delagi

Digital Equipment Corporation

To appear in:
Proceedings of the Fifth Annual Computer Science Symposium
University of South Carolina, Columbia, SC
April 7-8, 1989

Support for Fine-Grained Message Passing in Shared Memory Multiprocessors*

Gregory T. Byrd†
Stanford University

Bruce A. Delagi
Digital Equipment Corporation

Abstract

Since the highest performance parallel implementation of an application may require either a demand-driven (shared variable) or a data-driven (message passing) execution paradigm, we need to consider the efficient implementation of both in general purpose multiprocessor systems. Furthermore, as large-scale systems, utilizing hundreds or thousands of processors, are developed, it becomes increasingly desirable to support more finely grained execution than is feasible in current systems.

This paper proposes a hardware mechanism for supporting fine-grained messages on large-scale shared memory multiprocessor systems. The support is based on the concept of *streams* and is integrated with the cache and memory management system. We analyze the potential benefit of such support, and discuss an initial implementation, which will be simulated to provide more detailed performance information.

1 Introduction

There are two "traditional" approaches to parallel processing—shared variables and message passing—and these approaches have generally been considered incompatible. More recently, however, they have come to be viewed as extreme points along a continuum of communications styles [24]. Some applications are more naturally programmed in terms of messages, others in terms of shared variables, and still others can benefit from a hybrid approach.

The debate between messages and shared memory predates the widespread availability of parallel machines. Lauer and Needham [19] discuss the issue in the context of uniprocessor operating systems which support multiple threads of control. They compared a shared memory approach based on monitors with a message-oriented approach, and concluded that the two paradigms are essentially duals of one another—neither is intrinsically better, in terms of expressing concurrent

*This work was supported by Digital Equipment Corporation by DARPA Contract F30602-85-C-0012, by NASA Ames Contract NCC 2-220-S1, and by Boeing Contract W-266875. G. Byrd was supported by a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

†The authors may be reached at: Knowledge Systems Laboratory, Stanford University, 701 Welch Road, Bldg. C, Palo Alto, CA 94304

execution or in terms of high-level performance. Any differences in performance, they concluded, must come from features at the architectural level, not the operating system level.

Unfortunately, the architectural differences among current large-scale parallel computers are significant. Two major classes of architecture have emerged [illegible] designed to fundamentally support one of the two programming paradigms—shared variables [illegible] messages. Although either style of program can, of course, be executed, there may be a [illegible] performance penalty for using the style not directly supported by the hardware.

For example, most message passing machines [1,13,25,26] [illegible] a significant startup cost (hundreds of processor cycles [27]) for sending a message. Additionally, the memory on each processing node is not directly accessible by other processors, so that [illegible] data from a remote processing node requires message sending, reception, and interpretation [illegible] the requesting and responding processors.

Similarly, shared memory multiprocessors [2,21,22] provide low-overhead *demand-driven* access to global memory locations, which is the type of support needed for shared variables. However, they have very little support for more *data-driven* styles of communication, such as messages. Data-driven operations are typically implemented in software, either directly as application code, or through operating system routines (e.g., Mach [32]), or by handling interprocessor interrupts.

The effect of these added overheads is to increase the *granularity of computation* required for efficient execution. We define granularity as the amount of work performed by a *thread*[1] of computation before non-local communication is required, either through sending a message or accessing a shared memory location. For many applications, granularity naturally decreases as the number of processors increases—i.e., a fixed amount of work is divided among more and more threads. Therefore, decreasing the achievable granularity is especially desirable in large-scale systems utilizing hundreds or thousands of processors.

This paper proposes a hardware mechanism for supporting fine-grained message passing in large-scale shared memory multiprocessors. The mechanism is based on the concept of a *stream*, which represents a potentially infinite sequence of values. We use the memory management system to provide efficient communication of stream values, to synchronize instruction execution with the availability of such values, and to associate thread activation with the arrival of values on a stream.

In the next section, we will discuss why we choose to add message support to a shared memory multiprocessor, rather than adding shared variable support to a message passing machine. Then we will justify the use of special-purpose hardware by making lower-bounds performance estimates of various software implementations of messages. Finally, we describe the specifics of the proposed hardware.

[1] We use the term "thread," as in Birrell [3], to denote one of possibly many independent sequential flows of control within a single address space. The term "process" has come to be associated with a single flow of control associated one-to-one with an address space.

2 Why Shared Memory?

There are (at least) two approaches to creating a machine which has efficient, fine-grained support for both messages and shared variables:

- add support for shared variables to a message-passing machine, or
- add support for messages to a shared memory machine.

Two proposals which adopt the first approach are Dally's message-driven processor (MDP) [4,5] and Lindsay's shared memory hypercube [20]. Our proposal adopts the second approach.

2.1 MDP

The message-driven processor (MDP) [5] is designed to support fine-grained message passing between objects by providing *direct execution* of messages as they arrive from the network. The processor consists of two cooperating subunits: the message unit (MU), which accepts messages from the network, and the instruction unit (IU), which interprets the messages.

When a message arrives, the message unit either passes it directly to the instruction unit, if the IU is not currently busy, or buffers it in memory. (There is no cache in the proposed implementation of the MDP, but all the memory is on-chip, so a single-cycle memory access is achieved.) The instruction unit uses the first word of the message to branch to a subroutine for handling that type of message. The remaining words of the message are interpreted by the handler subroutine.

The MDP provides a *global namespace* for objects—all references to objects are in terms of a global identifier, which is then translated into a processor identifier and a physical address for that processor. A set-associative lookup mechanism is provided to assist in the translation process.

With low-latency message execution and a global namespace, the MDP could be used to execute shared variable programs. However, read and write requests must be interpreted by the instruction unit. This takes up cycles that could be used to run application code and is likely to be slower than servicing the request with a memory controller. Also, there is no hardware mechanism for maintaining consistency between copies of writeable objects. Either a software mechanism would be needed, or copies of writeable data would not be allowed.

2.2 Shared Memory Hypercube

Lindsay [20] proposes to support shared memory operations on a hypercube multicomputer by allowing the communications controller to directly interpret memory request messages. These messages could include the usual read and write operations, as well as more complex operations like fetch&add or swap.

Unlike the MDP approach, access to physical memory can be handled independently of the applications processor, but operations requiring virtual addresses and/or cache coherence could involve interrupting the processor. In comparison, our approach closely couples the network interface with the cache and virtual memory system—we can consider writing messages directly into

the cache or receiving virtual, instead of physical, addresses over the network without interrupting the processor. Additionally, automatic coherence of information cached by several processors is implicit.

2.3 Proposed Approach: Virtual Memory Support

In contrast to the above, we propose to integrate support for messages with the cache and virtual memory management system of a shared memory multiprocessor. This has the following advantages:

- Memory access requests are handled directly by the cache/memory controller, rather than by the processor, even if virtual addresses are passed over the network.
- The message passing mechanisms can be tightly integrated with existing cache and memory management hardware. Common mechanisms and resources can be used for handling messages, cache coherence operations, and block transfers.
- Stock processor and memory modules can be used. This lets us utilize the best uniprocessor technology, as it becomes available, with little or no redesign of the message passing mechanisms. It is also in keeping with what has been learned concerning the benefits of building fast, simple, general-purpose processors.
- The interface between the processor/compiler and the message passing hardware is based on reading and writing memory locations, and protection is provided by the normal virtual memory protection mechanisms.

3 Why Hardware Support?

In order to investigate the performance benefit of adding message passing hardware to shared memory multiprocessors, we estimate the performance achieved by various shared memory implementations of messages. We then compare those estimates with the estimated performance of a system with dedicated message-handling hardware.

3.1 Message Passing Implementations

In all the shared memory implementations, we assume the following operations:

1. The message is written into a buffer in shared memory.
2. The receiver is notified that the buffer contains a message.
3. The message is read from the buffer.
4. The sender is notified that the buffer is empty and available for a new message.

In this paper, we assume a single-sender, single-receiver (producer/consumer) style of message passing. Message buffers are preallocated (in both the shared memory and dedicated hardware implementations), and a single synchronization variable is used to indicate the full/empty status of the buffer.

We consider the following implementations:

- *Paracomputer:* This implementation, based on Schwartz's ideal *paracomputer* model [23], supposes that any number of processors can concurrently read or write a memory location in one cycle. Although unrealizable, this represents the best-case implementation based only on shared memory.

- *Remote, no cache:* Here we assume that the message buffer is located in a memory module which is local to neither the sending processor nor the receiving processor. This represents the worst case shared memory implementation, but it could arise in practice, either because there is no prior knowledge about the location of either the sender or receiver, or because of an architecture which does not provide local memory.

- *Receiver-local, no cache:* The message buffer is local to the receiving processor. The message is written a word at a time by the sending processor. The synchronization variable is also allocated local to the receiving processor.

- *Block transfer, no cache:* The message buffer is local to the sender; the receiver transfers the entire message into its local memory (using a block read) and then reads it locally.

- *Invalidate-based cache:* This implementation assumes that each processor has an infinitely large cache, which is kept coherent through some invalidate-based hardware coherence mechanism. Both the sending and receiving processors are assumed to have a copy of the buffer in their caches at the beginning of the message passing sequence.

- *Update-based cache:* This is the same as the previous case, except that an update-based coherence protocol is used.

- *Message coprocessor:* This implementation assumes the presence of dedicated message-handling hardware which reads and writes messages *directly from cache*. As in the shared memory implementations, a single variable is used to indicate the status of the buffer—this variable is local to the buffer and is only accessed by the message coprocessor and the receiving processor.

For each of these implementations, we estimate the latency of sending a single N-word message between two processors, with the following assumptions:

- *Cut-through network:* We assume a point-to-point interconnection network between processors, utilizing cut-through routing [6,18]. Each network channel is W bits wide, and the network may be clocked at a different rate than the processors. Each packet contains a t-bit target and L bits of data. For this analysis, every packet travels the average distance in the network, denoted as $\overline{D}$.

- *No contention:* There is no modelled contention, either in the network or at the memory modules.
- *Local memory:* We assume that some globally accessible memory module is local to each processor. The cost of accessing this local memory is M processor cycles. The memory is assumed to be interleaved, so that accessing a block of data takes M cycles for the first word and one cycle for each word thereafter.
- *Single-cycle cache:* For implementations which use cache, we charge one processor cycle for each cache access.
- *Multiple outstanding writes:* To maintain memory consistency, writes to shared memory locations must be acknowledged. We assume that multiple write requests may be outstanding—i.e., an arbitrary number of writes may be issued before the first is acknowledged. We enforce consistency by using *fence* or *delay* operations, which cause the processor to wait until all outstanding writes are acknowledged, before accessing synchronization variables. (See Shasha and Snir [29].)
- *Compiler prefetching:* We assume that the compiler can issue read requests to non-local memory locations before the data is actually needed—thus read requests can be pipelined. (This is a assumption which optimistically favors the shared memory implementations, since it may mean that there are enough free registers available to issue read requests for the entire message before the first word arrives.)
- *Cost-free directory access:* Since we are interested in scalable systems, we assume a directory-based cache coherence protocol, but directory operations are not counted in our performance estimates. In other words, we assume that each cache has instantaneous access to directory information. For example, it always "knows" which other caches have copies of shared data. Thus, our estimates are a lower bound on the actual time required by the cache-based implementations. (We estimate the error to be on the order of 10%.)

The latency equations derived from these assumptions, for each implementation, are given in the appendix.

3.2 A Concrete Example

The performance of each of the above implementations depends on the characteristics of the underlying hardware. In this section, we model a 256-processor torus-connected machine, representative of the "next-generation" scalable multiprocessors. The network parameters are modelled after MIT's proposed Jellybean machine [1.31]—eight-bit bidirectional channels, clocked at four times the processor clock rate. The parameters of the torus model are shown in table 1.

Figure 1 shows the estimated message latency for each of the implementations discussed earlier as the message size, N, increases from one to 32 words. (N refers to the size of the *data only*, not the entire network packet.) Figure 2 shows the same data, relative to the performance of the message coprocessor implementation.

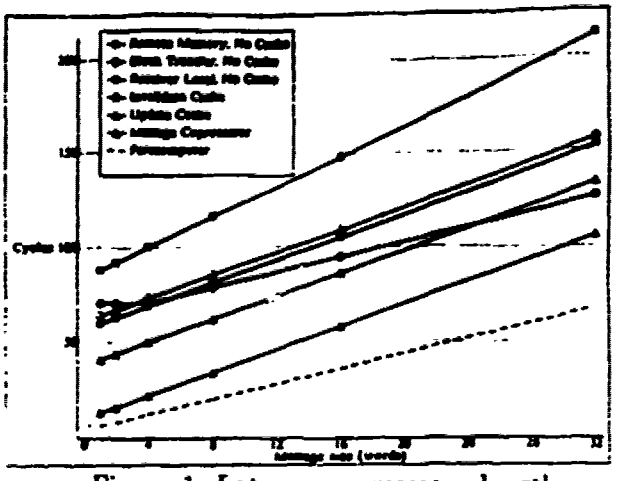


Figure 1: Latency *vs.* message length.

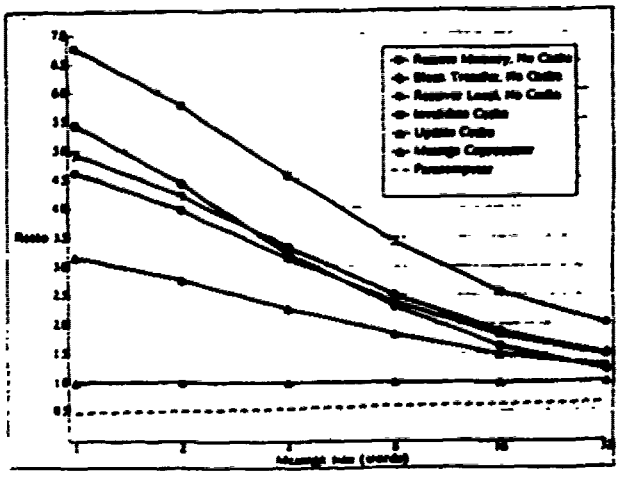


Figure 2: Relative latency *vs.* message length.

Aside from the ideal paracomputer implementation, the message coprocessor scheme shows the best performance over all message sizes. The update-based cache is next—for eight-word messages, it is about 80% worse than the message coprocessor. This is followed by receiver-local, invalidate-based cache, and block transfer, which are tightly grouped at around 2.4 times the latency of the message coprocessor (for eight words). Finally, the clear loser is the remote-memory based implementation, which has relative latency around 3.5 for eight-word messages and is, in fact, the slowest for all messages sizes.

The major noticeable trend in figure 2 is that the relative performances of the different schemes become closer as the message size increases. In fact, for 32-word messages every implementations is within a factor of two of the message coprocessor approach. This is because the data transfer cost dominates the synchronization cost for large packets. The data transfer costs differ mostly because of fixed-sized overheads, whose relative impact diminish at larger message sizes.

The next thing to notice is that, in figure 1, the curve for the invalidate-based cache has a smaller slope than the rest, showing that its performance is less dependent on message size than the other schemes. (The flat part of the curve is for message sizes less than or equal to the cache line size—four words.) This is because compiler prefetching and multiple outstanding writes allow

Parameter	*Symbol*	*Value*
Avg. network distance (hops)	D	8
Channel width (bits)	W	8
Cycle ratio (network/processor)	ρ	0.25
Local memory access (proc. cycles)	M	4
Cache line size (words)	B	4
Message size (words).	N	8

Table 1: Machine parameters for 256-processor torus.

most of the fetch and invalidation time to be overlapped with reading and writing from the cache.

Figures 3–8 show how varying the machine parameters affects the performance of the various alternatives, with the message size fixed at eight words.

The effects of changing the average distance ($\overline{D}$) are shown in figures 3 and 4. Decreasing distance corresponds to using a smaller (or higher-dimensioned) network or increasing the locality of the computations. High values of $\overline{D}$ correspond to large networks or poor locality. All of the shared memory implementations are more sensitive to distance than the message coprocessor case, since round trips through the network may be required for synchronization and data transfer. The remote-memory and invalidate-based cache implementations are the most strongly dependent on network distance—remote-memory, because both the sender and receiver must access data at distance $\overline{D}$, and invalidate-based cache, because reading the lock requires a cache line fetch, which means a round trip through the network.

The effects of changing the relative network and processor cycles times (ρ) are shown in figures 5 and 6. Slowing down the network or speeding up the processor would have the affect of *increasing* ρ. Changes in the relative network speed have dramatic effects on the absolute latencies of all the approaches (except the paracomputer, which does not depend on network latency). The block transfer and message coprocessor approaches are the best for networks with cycles times equal to or larger than the processor's, with equal channel width. These methods use long packets (rather than multiple small packets) to make efficient use of the network resources. As the network gets faster, the cache-based implementations show the greatest relative improvement, and, in fact, come very close to the messsage hardware approach.

In this model, increasing the channel width (W) has exactly the same effect as decreasing the cycle ratio (ρ)—i.e., speeding up the network. With reference to figures 5 and 6, increasing the width from 8 to 32 bits is the same as decreasing ρ from 1/4 to 1/16 (the left-most point on the graph). While it may seem easier than speeding up the network, note that increasing the channel width may have a significant cost impact, in terms of pin count, wiring costs, total area, and power.

Less interesting are the effects due to changing the local memory access time (M), shown in figure 7, and the cache line size (B), shown in figure 8. The local memory access only affects the non-cache implementations[2]: the block transfer and receiver-local schemes are essentially equivalent to an update-based cache for a single-cycle local memory, but become worse than either cache-based implementation for access times greater than four cycles.

Cache line size (figure 8) only affects the invalidate-based cache: the update-based cache sends every write to the network and no fetches are needed, so line size has no effect. In our model, small lines actually perform better, because reading the synchronization variable causes an entire line to be fetched. (This can be improved by using a *load-through* fetch [30], in which the referenced data is presented immediately to the processor, without waiting for the rest of the line.) Lines bigger than eight words (the message size), of course, cause latency to degrade dramatically, since more data than necessary is being transferred.

[2]Of course, finite caches would cause the other schemes to degrade with increasing local memory access times, due to local cache misses

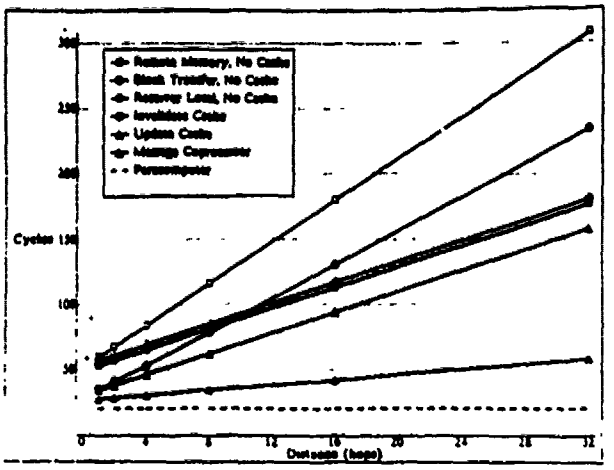


Figure 3: Latency *vs.* avg. distance.

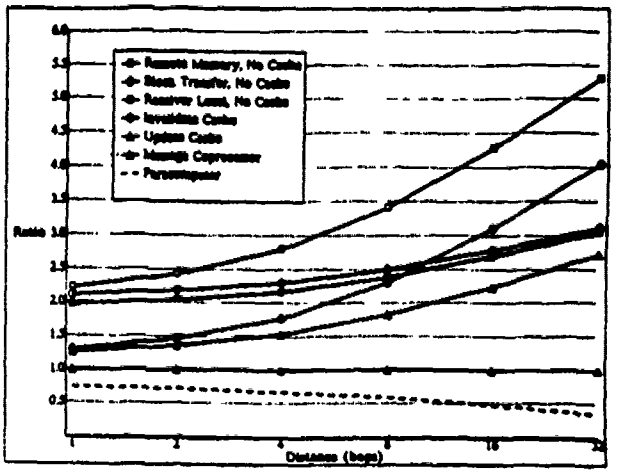


Figure 4: Relative latency *vs.* avg. distance.

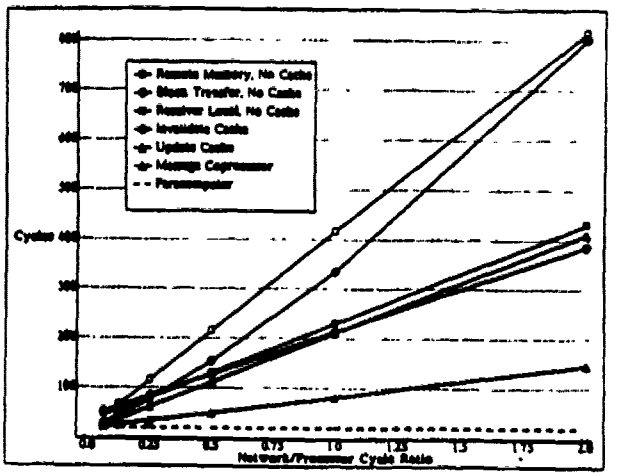


Figure 5: Latency *vs.* cycle ratio.

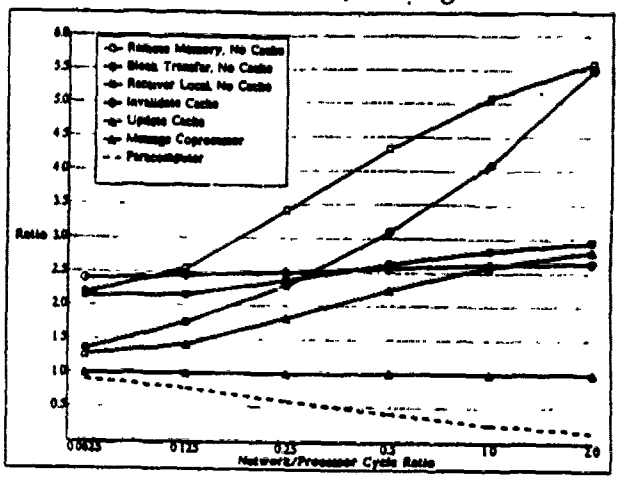


Figure 6: Relative latency *vs.* cycle ratio.

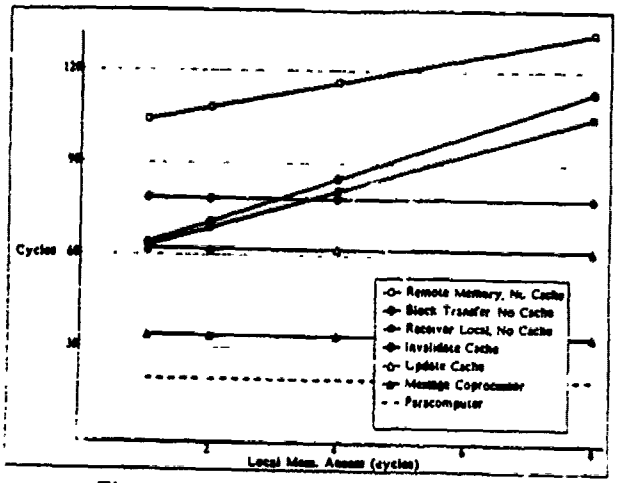


Figure 7: Latency *vs.* memory access

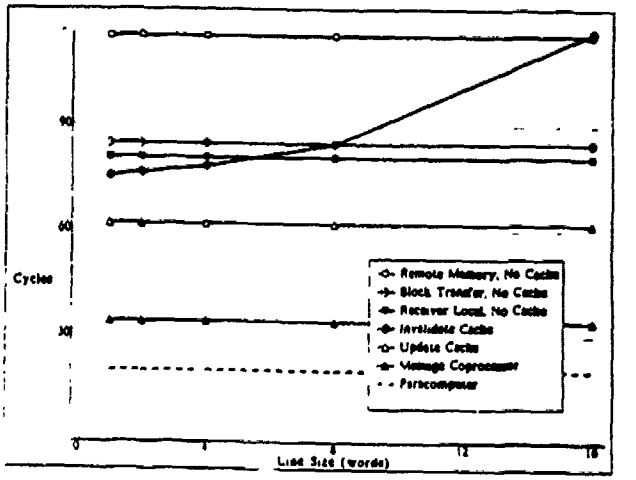


Figure 8 Latency *vs.* line size

3.3 Conclusions

The message coprocessor approach shows the best performance because:

- messages are read from and written to the cache directly, rather than to local memory;
- only one producer-driven network packet needs to be transmitted, as opposed to multiple memory request and acknowledgement packets required by the shared memory implementations; and
- local, fast synchronization is used to notify the receiver that a new message is present.

In the ideal case, a shared memory approach would be better, as shown by the performance of the paracomputer model, because the producer writes the message directly to the place where it is read by the receiver, and no copying of the message is needed. In the cases we have studied, however, the cost of multiple network transactions for data access and for synchronization can outweigh the cost of copying. Additionally, there is the cost of allocating and deallocating global message buffers, which is not considered in this cost model. The message coprocessor has an advantage in this area, as well, since allocation and deallocation can be done locally (as in the Ametek 2010 [26]).

Our performance estimates show that the shared memory implementations (except for remote memory) have a message latency of 1.5–2.5 times that of the message coprocessor for eight-word messages. Two important questions come to mind: Why worry about such small messages? And does a factor of two improvement warrant the additional hardware?

With respect to the first question, it is our expectation that large-scale systems will need to support more finely grained computation. Except for applications which can increase in *size* as the number of processors grows, yielding "size-up" or "scaled speedup" [11], most applications will leverage large systems by dividing work up into smaller chunks which can be performed in parallel. As the size of computational chunks decreases, so will the average message size.

We cannot answer the second question now: the answer will come from the investigations that we are now pursuing. However, we note that the factor of two estimate is based on a simple analytical model, which does not address the issue of network or memory contention. The shared memory implementations utilize more network transactions (to communicate the same amount of data) than the message coprocessor approach, so it is likely that network contention would cause their relative performance to further degrade. Also, the analytical model was based on several assumptions which optimistically favored the shared memory implementations. The cache-based implementations, for instance, would perform somewhat worse if directory operations were included. Finally, only a simple synchronization protocol, based on a single sender and receiver, was modelled. More complex protocols involving multiple senders and receivers require the management of global message queues—the message coprocessor approach, however, only requires local queue management. Even using the fetch&add-based queue management routines described in [10], we estimate that the relative message latencies for eight-word messages would increase to 3.5–6, rather than 1.5–2.5.

In summary, the achieved increase in performance by using special message-handling hardware appears to be at least a factor of two for small messages. The true benefit of supplying such

hardware, of course, depends on the cost of the hardware and the actual effect on overall performance. To more accurately assess the potential costs and benefits, we need to consider a specific implementation.

4 Proposed Hardware Support

In this section, we introduce a new hardware mechanism for supporting fine-grained message passing in shared memory multiprocessors. The mechanism is based on the concept of *streams*, described below, and is integrated with the cache and virtual memory system.

4.1 Streams

A *stream* is a data structure which represents a potentially infinite sequence of values communicated between processes. The values may be self-referential entities (such as integers, strings, or arrays), pointers to global data, or pointers to other streams.

Streams are produced and consumed incrementally, and operations on streams are *non-strict*—that is, the operations can be applied to the stream before the entire sequence of values has been computed. Producers write to a stream, which causes a value to be placed at the end of the sequence. The order in which values appear on the stream does not necessarily correspond to the order in which they were written. Also, writes from multiple producers are merged non-deterministically. Consumers read from a stream, which returns the first value in the sequence and (optionally) removes it from the sequence.

In addition to communication, a stream also provides a means of synchronization among processes. Upon reading an empty stream, a consumer may become blocked and may later be resumed when a value is written to the stream.

Streams may be considered a generalization of *futures* [12]. While an unsatisfied future represents a "promise" for a single value, an empty stream represents the promise for (the next item in) a sequence of values. In fact, a stream may be implemented as a sequence of futures—each future being evaluated as pair which contains a value and a future which represents the remaining values. (This is similar to the representation of streams in Concurrent Prolog [28] as sequences of logic variables.)

We choose streams, rather than futures, as the primitive to be supported for these reasons:

- We expect processes which communicate to continue communicating over time (as in [8]). We therefore prefer to reuse a stream rather than newly allocate a (write-once) future for each individual communication.
- Multiple writers can more easily be supported by a direct implementation of a stream, compared to a sequence of futures, in which each writer must specify the location of the next item. In other words, an extra *merge* operator is not required for non-deterministic operation.
- Often, a multi-word item needs to be communicated (e.g., a task identifier and a couple of arguments). While a single-word future would required that a pointer to the item be written,

or that several futures be satisfied, a stream can directly store multiple words without the need for indirection.

Stream-like constructs have been used in several concurrent environments—such as *pipes* in the V system [34], or *mailboxes* in the Spur Lisp environment [33]. Also, streams are an integral part of Lanzina [9], a concurrent object-oriented extension of Lisp.

4.2 Stream Pages

We implement a stream as a special type of virtual memory page, the *stream page*. Stream pages are global, but only locally cacheable—a processor may directly access (and cache) only those streams which are assigned to its local memory.

In addition to the usual status and usage information maintained by the virtual memory system, stream pages must manage the following:

- **head**: a pointer (modulo the page size) to the first valid data word in the page;
- **tail**: a pointer (modulo the page size) to the next available data word in the page;
- **empty**: a flag which, when set, indicates that the stream contains no valid data;
- **overflow**: a flag which is set when the number of words on the stream exceeds the number of words in a stream page;
- **blocked**: a flag which, when set, indicates that there are threads waiting for data to arrive on this (empty) stream;
- **threads**: a counter of the number of threads waiting on a blocked stream;
- **read_lock**: a flag which, when set, indicates that a thread (or the network) is currently removing data from the stream;
- **write_lock**: a flag which, when set, indicates that a thread (or the network) is currently adding data to the stream;
- **linked**: a flag which, when set, indicates that this stream is linked to another stream;
- **link**: a pointer (virtual address) to another stream;
- **trap**: a flag which, when set, indicates that an error condition should be signalled when reading an empty stream, rather than stalling the processor.

As described earlier, a stream represents a sequence of values. This sequence is implemented in the memory system by treating the stream page as a circular buffer. The **head** and **tail** denote which data words in the page are valid stream entries. Any attempt to read an address outside this range results in an error.

The above control information may be stored in the first few words of the stream page (or in some other location associated with the page table entry for the stream page). During active use, however, this information is needed for every stream access. Therefore we provide a special hardware module, called the *stream translation unit (STU)*, which holds the data for the currently active streams. This is analogous to the more traditional translation lookaside buffer (TLB), which contains the most recently used virtual-to-physical address translations. As with a TLB, a reference to a stream which is not contained in the STU results in a *miss*, which causes the control information for the referenced stream to be loaded into the STU for later use. (We still must use a TLB to perform virtual-to-physical translation for "normal" memory operations.)

4.2.1 Reading a stream

Currently, we only allow a thread to read from a stream which is allocated to its own local memory.[3] To read from a stream, the thread first reads the **head**. Reading the **head** sets the **read_lock** and returns the current **head** and the old value of **read_lock**, so the thread can tell whether or not the lock has been required. Any attempt to read data from a locked stream results in an error. Locking the stream is required since the reading thread may be pre-empted or descheduled for a indefinite period of time. During that time, some other thread may be scheduled which wishes to read the same stream—the result is that each thread may read half a message.

If the **empty** flag is set during a read operation, then there are two possible actions:

1. If the **trap** flag is clear (or if the **write_lock** is set), then the processor stalls until data is written to the stream. (There may be a timeout interrupt that prevents the processor from stalling indefinitely.)

2. If the **trap** flag is set, then the **blocked** and **write_lock** flags are set, and an error condition is returned. The error handler should then write a pointer to the thread's control block on the tail of the stream and increment the **threads** counter—when data arrives, the thread will be rescheduled. In the meantime, another thread may execute.

4.2.2 Writing to a stream

A message is never written directly to the stream from which it is read. Instead, the message is written to an intermediate, write-only stream (called the *source* stream) which is linked to the desired destination stream (called the *sink*). The source stream is always local to the writer—when the message is completely written, the stream-handling hardware copies it (possibly across the network) to the sink stream, which is specified by the source stream's **link** field.[4]

There are two reasons for adopting this indirect approach:

[3] A scheme to allow read access to remote streams is under development. Remote stream access could be very useful in implementing efficient shared queues, such as task queues.

[4] The **linked** flag is used to indicate whether the link field is valid.

- **Having multiple, non-local writers would require providing a consistent view of the streams head and tail. Instead, our approach has each writer maintain its own local stream information.**

- **A local writer can be pre-empted or swapped out indefinitely. If this thread were writing directly to a sink stream, then the write lock would be held indefinitely, which would block any network messages arriving for that stream. Blocking the network for long periods of time can degrade the performance of the entire system, so it was decided that local threads should not write to the same stream as the network.**

Since several sources could be sending messages to the same sink, there is, of course, the possibility of overflow. We can try to avoid overflow by sending *negative acknowledge (NAK)* messages back to a source stream when the sink is almost full. The NAK would prohibit writes on the source until an ACK message arrives later, when the sink is not as full. When overflow does occur, however, we plan to reroute the message to an *overflow stream*, managed by the operating system, which would later copy the message to the proper stream. When the overflow stream overflows, the processor would have to be interrupted, in order to allocate a new overflow stream and/or service some of the overflow messages.

When a message is written to a stream which is blocked, then the waiting threads must be rescheduled. One way to accomplish this quickly is:

1. The first thread pointer on the blocked stream (see above) is transferred to a scheduling stream, which is serviced by the operating system.

2. When the thread is awakened, it removes the number of thread pointers indicated by the **threads** counter, and places them on the scheduling stream.

3. Finally, the awakened thread continues execution, probably by reading the message that arrived on the stream.

This approach places some burden on the awakening thread (or the scheduling routine), but allows an incoming message to be placed on the stream without waiting for an arbitrary number of threads to be placed on the scheduling stream.

4.3 Further Work

These ideas and mechanisms for hardware support of streams are preliminary and much more work needs to be done. We are in the process of refining the mechanisms and nailing down the details of a possible implementation. We are very interested in investigating the support of further stream operations, such as access to remote streams and a more extensive mechanism for linking streams (to provide a multicast-like facility). Some other ideas about useful stream operations can be found in the description of Lamina, a concurrent object-oriented language [9].

Once a reasonable implementation has been detailed, we must evaluate its cost and performance. The analytical techniques used at the beginning of this paper can only provide a crude estimate

of performance under (unreasonable) best-case conditions. In order to examine contention effects and more complex data transfer patterns, we will simulate a target implementation running test applications.

In particular, we will use the CARE simulation system [7], which allows us to run non-trivial applications on a variety of multiprocessor configurations. CARE also provides a flexible, non-intrusive instrumentation system, which will enable us to measure various aspects of system performance, such as message latency.

4.4 Related Work

The BBN Butterfly [2] is large-scale shared memory machine which also provides some microcode support for message passing. A coprocessor, called the *processor node controller*, is responsible for interfacing the processor and memory to the interconnection network, and contains microcode routines for atomic operations, block transfers, and so forth.

In particular the microcode supports operating system constructs called *events* and *dual-queues*. Events are used to signal a process—a posted event may cause a blocked process to become unblocked. (Portions of the process scheduler are also implemented in microcode.) Dual-queues are used to hold either one-word data items (usually pointers) or events which represent processes waiting for data to be placed on the queue.

While some aspects of events and dual-queues are similar to the stream approach that we have proposed there are significant differences:

- Streams can accept multiple-word items. This means that the allocation and deallocation of buffers is automatic, for messages smaller than the stream page size. Dual-queues only accept events or one-word data items.
- Our stream support is integrated with support for coherent caches. The Butterfly does not provide caches.
- The microcode routines on the node controller are accessed by passing a pointer to a control block to a special memory address [17]. Streams are accessed merely by reading or writing to the proper page—all the necessary control information is managed by the hardware. The Butterfly interface is more flexible but (potentially) more expensive. Our approach is aimed toward providing single-cycle service for the most critical operations; we rely on exception-handling software to handle the more complex cases.

5 Summary and Conclusions

In summary, it is our conjecture that future large-scale multiprocessor systems should support both fine-grained messages and fine-grained access to shared data. Fine-grained messages implies efficient communication of small blocks of data between threads, synchronization of instruction execution with the availability of message data, and low-overhead mechanisms for associating thread

activation with message arrival. Fine-grained shared variables implies direct servicing of memory access requests by the memory system (without the involvement of the processor) and (probably) a hardware-based mechanism for maintaining cache coherence.

The approach that we propose is to integrate support for messages with the memory management system of a shared memory multiprocessor. We have presented evidence, in the form of message latency estimates, that suggests that fine-grained messages can benefit from dedicated hardware support.

Finally, in order to better quantify the cost and benefits of such hardware, we proposed a possible implementation, based on streams. A model of this implementation will be simulated, running benchmark applications, so that performance parameters such as message latency, memory traffic, and network throughput can be measured. Based on these measurements, we can evaluate the impact of the added hardware on the overall system performance.

A Latency Derivations

In this appendix, we derive the latency equations used to estimate performance in section 3. We are interested in the best-case latency of sending an N-word message between processes A and B, each on a different processor.

For each implementation, the latency of each portion of the message transfer is identified. Since these actions are serialized, the total latency is computed by adding the latencies of each step.

When either process accesses a lock (i.e., synchronization variable), only the portion of the access that is serialized with the rest of the message transfer is counted. For example, we assume that the reading of a lock happens immediately following an associated write operation—thus we do not charge for the network delay of the read request or the write acknowledge, since they may be overlapped with other portions of the transaction.

Other assumptions are explained in section 3.1.

A.1 Terminology

$\overline{D}$	=	average network distance (hops)
t	=	size of target address (bits)
W	=	network channel width (bits)
$\mathcal{T}(k)$	=	network cycles to transmit k words $= \lceil \frac{t}{W} \rceil \overline{D} + \lceil \frac{32k}{W} \rceil$
$\mathcal{S}(k)$	=	network cycles to place k words onto the network $= \lceil \frac{t}{W} \rceil + \lceil \frac{32k}{W} \rceil$
ρ	=	ratio of network cycle time to processor cycle time
M	=	memory access time, in processor cycles
B	=	cache line size, both for transfer and coherency

A.2 Paracomputer

1. A reads lock: 1 cycle;
2. A writes data: N cycles;
3. A writes lock: 1 cycle;

4. **B reads lock: 1 cycle;**
5. **B reads data: N cycles;**
6. **B writes lock: 1 cycle.**

A.3 Remote Memory, No Cache

1. A reads lock: $\rho\mathcal{T}(1) + M$ cycles;
2. A writes data: $(N-1)\rho\mathcal{S}(1) + 2\rho\mathcal{T}(1) + M$ cycles;
3. A writes lock: $\rho\mathcal{T}(1) + M$ cycles;
4. B reads lock: $\rho\mathcal{T}(1) + M$ cycles;
5. B reads data: $(N-1)\rho\mathcal{S}(1) + 2\rho\mathcal{T}(1) + M$ cycles;
6. B writes lock: $\rho\mathcal{T}(1) + M$ cycles.

A.4 Receiver-Local Memory, No Cache

1. A reads lock: $\rho\mathcal{T}(1) + M$ cycles;
2. A writes data: $(N-1)\rho\mathcal{S}(1) + 2\rho\mathcal{T}(1) + M$ cycles;
3. A writes lock: $\rho\mathcal{T}(1) + M$ cycles;
4. B reads lock: M cycles;
5. B reads data: $M + (N-1)$ cycles;
6. B writes lock: M cycles.

A.5 Sender-Local Memory, Block Transfer

1. A reads lock: $\rho\mathcal{T}(1) + M$ cycles;
2. A writes data: $M + (N-1)$ cycles;
3. A writes lock: $\rho\mathcal{T}(1) + M$ cycles;
4. B reads lock: M cycles;
5. B transfers data: $\rho\mathcal{T}(1) + M + \rho\mathcal{T}(N)$ cycles;
6. B reads data: $M + (N-1)$ cycles;
7. B writes lock: M cycles.

A.6 Invalidate-Based Cache

1. A reads lock (from B's cache): $\rho(\mathcal{T}(1) + \mathcal{T}(B))$ cycles;
2. A writes data: $(\lceil \frac{N}{B} \rceil - 1)B + \max(2\rho\mathcal{T}(1) + 1, B)$ cycles[5];
3. A writes lock: $\rho\mathcal{T}(1) + 1$ cycles;
4. B reads lock (from A's cache): $\rho(\mathcal{T}(1) + \mathcal{T}(B))$ cycles;
5. B reads data (from A's cache): $(\lceil \frac{N}{B} \rceil - 1)B + \rho(\mathcal{T}(1) + \mathcal{T}(B))$ cycles;
6. B writes lock: $\rho\mathcal{T}(1) + 1$ cycles.

A.7 Update-Based Cache

1. A reads lock: 1 cycle;
2. A writes data: $(N-1)\rho\mathcal{S}(1) + 2\rho\mathcal{T}(1) + 1$ cycles;
3. A writes lock: $\rho\mathcal{T}(1) + 1$ cycles;
4. B reads lock: 1 cycle;
5. B reads data: N cycles;
6. B writes lock: $\rho\mathcal{T}(1) + 1$ cycles.

A.8 Message Coprocessor

1. A writes data: N cycles;
2. data is transmitted: $\rho\mathcal{T}(N)$ cycles;
3. B is notified: 1 cycle;
4. B reads data: N cycles;
5. B frees buffer: 1 cycle.

References

[1] Ramune Arlauskas. iPSC/2 System: a second generation hypercube. In volume 1 of *Hypercube '88* [14], pages 38–42.

[2] BBN Laboratories Incorporated. Butterfly parallel processor overview. BBN Report 6148. March 1986.

[3] Andrew D. Birrell. An introduction to programming with threads. SRC Research Report 35. Digital Equipment Corporation. January 1989.

[5] Depending on the line size, B, the last acknowledgement might have returned before the last of the line is written.

[4] William J. Dally. Fine-grain message-passing concurrent computers. In volume 1 of *Hypercube '88* [14], pages 2–12.

[5] William J. Dally et al. Architecture of a message-driven processor. In *14th Annual Symposium on Computer Architecture*, pages 189–196. ACM, June 1987.

[6] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[7] Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. Instrumented architectural simulation. In volume 1 of *ICS '88* [16], pages 8–11.

[8] Bruce A. Delagi and Nakul P. Saraiya. ELINT in LAMINA: Application of a concurrent object language. Technical Report KSL-88-33, Knowledge Systems Laboratory, Stanford University, 1988. To appear in *SIGPLAN Notices*, April 1989.

[9] Bruce A. Delagi, Nakul P. Saraiya, and Gregory T. Byrd. LAMINA: CARE applications interface. In volume 1 of *ICS '88* [16], pages 12–21.

[10] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[11] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4), July 1988.

[12] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1984.

[13] John P. Hayes, Trevor Mudge, Quentin F. Stout, Stephen Colley, and John Palmer. A microprocessor-based hypercube supercomputer. *IEEE Micro*, pages 6–17. October 1986.

[14] *The Third Conference on Hypercube Concurrent Computers and Applications*. Pasadena, CA, January 1988. ACM Press, New York.

[15] *Proceedings of the 1988 International Conference on Parallel Processing*. The Pennsylvania State University Press, August 1988.

[16] *Proceedings of the Third International Conference on Supercomputing*. Boston, MA, May 1988. International Supercomputing Institute, Inc., St. Petersburg, FL.

[17] BBN Laboratories Incorporated. Development of a voice funnel system: Quarterly technical report no. 12. BBN Report 4845, January 1982.

[18] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267, 1979.

[19] **Hugh C. Lauer and Roger M. Needham.** On the duality of operating system structures. In *Proceedings of the Second International Symposium on Operating Systems.* IRIA, October 1978. Reprinted in *Operating Systems Review*, Vol. 13, No. 2, April 1979, pp. 3-19.

[20] Donald C. Lindsay. Towards a shared memory hypercube. Technical Report CMU-CS-88-190, Dept. of Computer Science, Carnegie Mellon University, November 1988.

[21] Tom Lovett and Shreekant Thakkar. The Symmetry multiprocessor system. In volume 1 of *ICPP '88* [15], pages 303–310.

[22] G. F. Pfister et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771.

[23] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, pages 484–521, 1980.

[24] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Design rationale for Psyche, a general-purpose multiprocessor operating system. In volume 2 of *ICPP '88* [15], pages 255–262.

[25] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[26] Charles L. Seitz et al. The architecture and programming of the Ametek Series 2010 multicomputer. In volume 1 of *Hypercube '88* [14], pages 33–36.

[27] Charles L. Seitz, Jakov Seizovic, and Wen-King Su. The C programmer's abbreviated guide to multicomputer programming. Technical Report Caltech-CS-TR-88-1, Department of Computer Science, California Institute of Technology, January 1988.

[28] Ehud Shapiro, editor. *Concurrent Prolog.* MIT Press, Cambridge, MA, 1987.

[29] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. Technical Report RC 12936 (#58037). IBM T. J. Watson Research Center, July 1987.

[30] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):173–530, September 1982.

[31] Paul Y. Song. Design of a network for concurrent message passing systems. Master's thesis, Massachusetts Institute of Technology, May 1988.

[32] Michael Young et al. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th Symposium on Operating Systems Principles.* ACM, November 1987.

[33] Benjamin Zorn et al. Features for multiprocessing in SPUR Lisp. Technical Report UCB/CSD 88/406, Computer Science Division (EECS), University of California, Berkeley, March 1988.

[34] Willy Zwaenepoel. *Message Passing on a Local Network.* PhD thesis, Stanford University, 1985.

Knowledge Systems Laboratory
Report No. KSL 86-14

March, 1986

CAREL: A Visible Distributed Lisp

by

Byron Davies

KNOWLEDGE SYSTEMS LABORATORY
Departments of Medical and Computer Science
Stanford University
Stanford, California 94305

CAREL: A Visible Distributed Lisp

Byron Davies

Knowledge Systems Laboratory
and Center for Integrated Systems
Stanford University
Palo Alto, California

and

Corporate Computer Science Center
Texas Instruments
Dallas, Texas

The author gratefully acknowledges the support of the following funding agencies for this project: DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing, under contract number W-260875.

Abstract

CAREL is a Lisp implementation designed to be a high-level interactive systems programming language for a distributed-memory multiprocessor. CAREL insulates the user from the machine language of the multiprocessor architecture, but still makes it possible for the user to specify explicitly the assignment of tasks to processors in the multiprocessor network. CAREL has been implemented to run on a TI Explorer Lisp machine using Stanford's CARE multiprocessor simulator [Delagi 85].

CAREL is more than a language: real-time graphical displays provided by the CARE simulator make CAREL a novel graphical programming environment for distributed computing. CAREL enables the user to create programs interactively and then watch them run on a network of simulated processors. As a CAREL program executes, the CARE simulator graphically displays the activity of the processors and the transmission of data through the network. Using this capability, CAREL has demonstrated its utility as an educational tool for multiprocessor computing.

1. Context

CAREL was developed within the Advanced Architectures Project of the Stanford Knowledge Systems Laboratory. The goal of the Advanced Architectures Project is to make knowledge-based programs run much faster on multiple processors than on one processor. Knowledge-based programs place different demands on a computing system than do programs for numerical computation. Indeed, multiprocessor implementations of expert systems will undoubtedly require specialized software and hardware architectures for efficient execution. The Advanced Architectures Project is performing experiments to understand the potential concurrency in signal understanding systems, and is developing specialized architectures to exploit this concurrency.

The project is organized according to a number of abstraction layers, as shown in Figure 1-1. Much of the work of the project consists of designing and implementing languages to span the semantic gap between the applications layer and the hardware architecture.

The design and implementation of CAREL depends mainly on the hardware architecture level. The other levels will be ignored in this summary, but are described briefly in the full paper. At the hardware level, the project is concentrating on a class of multiprocessor architectures. The class is roughly defined as MIMD, large grain, locally-connected, distributed

Layer	Research Question
Applications	Where is the potential concurrency in signal understanding tasks?
Problem-solving frameworks	How do we maximize useful concurrency and minimize serialization in problem-solving architectures?
Knowledge-representation and inference	How do we develop knowledge representations to maximize parallelism in inference and search?
Systems programming language	How can a general-purpose symbolic programming language support concurrency and help map multi-task programs onto a distributed-memory multiprocessor?
Hardware architecture	What multiprocessor architecture best supports the concurrency in signal understanding tasks?

Figure 1-1: Multiple layers in implementing signal understanding expert systems on multiprocessor hardware

memory multiprocessors communicating via buffered messages. This class was chosen to match the needs of large-scale parallel symbolic computing with the constraints imposed by the desire for VLSI implementation and replication. Like the FAIM-1 project [Davis and Robison 85], we consider each processing node to have significant processing and communication capability as well as a reasonable amount of memory - about as much as can be included on a single integrated circuit (currently a fraction of a megabit, but several megabits within a few years). Each processor can support many processes. As the project progresses, the detailed design of the hardware architecture will be modified to support the needs of the application as both application and architecture are better understood.

The hardware architecture level is implemented as a simulation running on a (uniprocessor) Lisp machine. The simulator, called CARE for "Concurrent ARray Emulator" (*sic*), carries out the operation of the architecture at a level sufficiently detailed to capture both instruction run times and communication overhead and latency. The CARE simulator has a programmable instrumentation facility which permits the user to attach "probes" to any object or collection of objects in the simulation, and to display the data and historical summaries on "instruments" on the Lisp machine screen. Indeed, the display of the processor grid itself is one such instrument.

2. Introduction

The CAREL (for CARE Lisp) language is a distributed-memory variant of QLAMBDA [Gabriel and McCarthy 84] and an extension of a Scheme subset [Abelson and Sussman 85]. CAREL supports futures (like Multilisp [Halstead 84]), truly parallel LET binding (like QLAMBDA), programmer or automatic specification of locality of computations (like Par-Alfl [Hudak and Smith 86] or Concurrent Prolog [Shapiro 84]), and both static assignment of process to processor and dynamic spread of recursive computations through the network via remote function call. Despite the length of this list of capabilities, CAREL is perhaps best described as a high-level systems programming language for distributed-memory multiprocessor computing.

The CAREL environment provides both accessibility and visibility. CAREL is accessible because, being a Lisp, it is an interactive and interpreted language. The user may type in expressions directly and have them evaluated immediately, or load CAREL programs from files. If the multiprocessing features are ignored, using CAREL is just using Scheme. The multiprocessing extensions in CAREL are derived from those of QLAMBDA. For example, PARALLEL-LET is a simple extension of LET which computes the values for the LET-bindings concurrently, at locations specified by the programmer or determined automatically.

CAREL gains its visibility through the CARE simulator; CAREL programmers can watch their programs execute on a graphic display of the multiprocessor architecture. Figure 5-1 shows CARE and CAREL with a typical six-by-six grid of processors. A second window on the Lisp machine screen is used as the CAREL listener, where programs are entered. As a CAREL program runs, the simulator illuminates each active processor and each active communication link. The user may quickly gain an understanding of the processor usage and information flow in distributed CAREL programs. CARE instruments may also be used to gather instantaneous and historical data about the execution of CAREL programs.

The rest of the paper is divided into a discussion of the philosophy of CAREL, a description of the language CAREL, and some illustrated examples of CAREL in action on the CARE simulator.

3. Philosophy and Design

The CAREL language was developed with a number of assumptions in mind. The following assumptions are stated very briefly for this summary but appear in expanded form in the full paper:

1. CAREL (like Multilisp) was designed to augment a serial Lisp with "discretionary" concurrency: the programmer, rather than the compiler or the run-time support system, decides what parts of a program will be concurrent. CAREL provides parallelism through both lexical elaboration and explicit processes [Filman and Friedman 84].

2. Similarly, CAREL was designed to provide discretionary locality: the programmer also decides *where* concurrent routines will be run. A variety of abstract mechanisms are provided to express locality in terms of direction or distance or both.

3. CAREL generally implements *eager* evaluation: when a task is created, it is immediately started running, even if the result is not needed immediately. When the result is needed by a strict operator, the currently running task blocks until the result is available.

4. CAREL is designed to automatically manage the transfer of data, including structures, between processors. CAREL supports general methods to copy lists and structures from one processor to another, and specialized methods to copy programs and environments.

5. CAREL is designed to maintain "architectural fidelity": all communication of both data and executable code is explicitly handled by the simulator so that all costs of communication may be accounted for.

6. CAREL provides certain specialized "soft architectures", such as pipelines, overlayed on the processor network.

7. Through CARE, CAREL graphically displays the runtime behavior of executing programs.

8. Finally, and unfortunately, CAREL ignores resource management, including the problem of garbage collecting data and processes on multiple processors. Resource management is a very important problem, but CAREL doesn't yet have a solution for it. CAREL currently depends on the memory management of the Lisp machine on which it

4. The Language

This section presents a language description of CAREL and examples - with graphics - of its use. The functions and special forms of CAREL were selected roughly as the union of the capabilities of QLAMBDA (as extended for distributed memory) and Par-Alfl. There has been no attempt as yet to create a minimal but complete subset of CAREL.

On top of Scheme subset, CAREL supports the following functions and special forms:

PARALLEL-LET: a special form for parallel evaluation of LET binding. Optionally, the programmer may specify the locations at which the values for binding are to be evaluated.

PARALLEL-LAMBDA: a special form to create asynchronously running closures. Optionally, the programmer may specify the location where the closure is to reside. The closure may also include state variables so that it's behavior may vary over time.

PARALLEL: a parallel PROGN, evaluating the component forms concurrently.

PARALLEL-MAP: a parallel mapping function which applies a single function to multiple arguments at multiple locations, returning a list of the results.

MULTICAST-MAP: a parallel mapping function which evaluates the same form at multiple locations and gathers up the values returned in the order in which they are returned.

FUTURE: a special form specifying a form to be evaluated and the site at which the evaluation should take place. Returns a future encapsulating the value that will eventually be returned.

TOUCH/FORCE: a function to force a future to give up its value.

ON: evaluates a form at a specified location. Equivalent to (TOUCH (FUTURE ...)).

8. Finally, and unfortunately, CAREL ignores resource management, including the problem of garbage collecting data and processes on multiple processors. Resource management is a very important problem, but CAREL doesn't yet have a solution for it. CAREL currently depends on the memory management of the Lisp machine on which it

4. The Language

This section presents a language description of CAREL and examples - with graphics - of its use. The functions and special forms of CAREL were selected roughly as the union of the capabilities of QLAMBDA (as extended for distributed memory) and Par-Alfl. There has been no attempt as yet to create a minimal but complete subset of CAREL.

On top of Scheme subset, CAREL supports the following functions and special forms:

PARALLEL-LET: a special form for parallel evaluation of LET binding. Optionally, the programmer may specify the locations at which the values for binding are to be evaluated.

PARALLEL-LAMBDA: a special form to create asynchronously running closures. Optionally, the programmer may specify the location where the closure is to reside. The closure may also include state variables so that it's behavior may vary over time.

PARALLEL: a parallel PROGN, evaluating the component forms concurrently.

PARALLEL-MAP: a parallel mapping function which applies a single function to multiple arguments at multiple locations, returning a list of the results.

MULTICAST-MAP: a parallel mapping function which evaluates the same form at multiple locations and gathers up the values returned in the order in which they are returned.

FUTURE: a special form specifying a form to be evaluated and the site at which the evaluation should take place. Returns a future encapsulating the value that will eventually be returned.

TOUCH/FORCE: a function to force a future to give up its value.

ON: evaluates a form at a specified location. Equivalent to (TOUCH (FUTURE ...)).

Evaluating a PARALLEL-LAMBDA sets up a closure at a remote site specified by *location* and returns a function of the specified arguments. When this function is applied, the list of evaluated arguments is sent to the remote closure, the remote evaluation is initiated, and a future is immediately returned. The remote closure created by PARALLEL-LAMBDA contains some state variables, bound in *state-bindings*. A state variable is changed by applying the PARALLEL-LAMBDA function to the arguments (:SET *variable-name value*).

parallel? is used, as in PARALLEL-LET, to determine whether parallelism is actually employed.

PARALLEL:

(PARALLEL . *body*)

The PARALLEL special form initiates the concurrent evaluation of the forms in the *body*. Control returns from PARALLEL when all of the forms have been evaluated. The value returned by PARALLEL is undefined.

PARALLEL-MAP:

(PARALLEL-MAP *function-form arguments-form locations-form*)

function-form evaluates to a function of one argument

arguments-form evaluates to a list, each member of which is to be used as an argument to the function

locations-form, evaluated to a list of locations.

PARALLEL-MAP, like MAP, applies a function repeatedly to arguments drawn from a list and returns a list of results. Unlike MAP, PARALLEL-MAP performs the function applications remotely, and returns a list of futures that will eventually evaluate to the results.

MULTICAST-MAP:

(MULTICAST-MAP *function-form locations-form*)

MULTICAST-MAP invokes a function of no arguments at each location in a list of locations. MULTICAST-MAP immediately returns a list of futures corresponding to the values that will eventually be returned. Since the function called takes no arguments, the values returned can be different only if they depend on the local state of the processor at the location of evaluation, as embodied in the "global" environment of that processor.

MULTICAST-MAP-NO-REPLY:

(MULTICAST-MAP-NO-REPLY *function-form locations-form*)

MULTICAST-MAP-NO-REPLY invokes a function of no arguments at each location in a list, but does not cause results to be returned. The value returned by MULTICAST-MAP-NO-REPLY is undefined.

PIPELINE:

(PIPELINE *stage1 ... stagen*)

where a stage is:

(*name args location-form state-variables . output-forms*)

For each stage expression, PIPELINE establishes a remote-closure at the specified location, and then links the remote closures so that the output of one stage becomes the input of the next stage. The linked closures form the working part of the pipeline. PIPELINE then returns a function which, when applied, passes its arguments on to the first stage of the pipeline and immediately returns a future which will eventually contain the result that comes out of the pipeline. To ensure that the results that comes out of the pipeline correspond one-for-one with the sets of arguments that went in, the future-object to hold the result is created atomically with the entry of the arguments into the pipeline and is passed along with the data through the pipeline.

5. Some Examples

PARALLEL-LET:

```
;;; This subroutine concurrently performs trivial computations at the four
;;; corner neighbors of a given location and collects the results.
;;;
(define (cycle-corners-1 start-location)
  (parallel-let t ((x1 (list 1 2) (neighbor 0 start-location))
                   (x2 (list 3 4) (neighbor 2 (neighbor 1 start-location)))
                   (x3 (list 5 6) (neighbor 3 start-location))
                   (x4 (list 7 8) (neighbor 5 (neighbor 4 start-location))))
    (append x1 x2 x3 x4)))

;;; CYCLE calls the subroutine starting at the current processor
;;;
(define (cycle) (cycle-corners-1 *here*))
```

PARALLEL-MAP (see Figure 5-1):

```
;;; FOUR-CYCLE calls the CYCLE program at four different locations
;;; in the processor grid.
;;;
(define (four-cycle)
  (parallel-map cycle-corners-1
                '((2 5) (5 2) (2 2) (5 5))
                '((2 5) (5 2) (2 2) (5 5))))
```

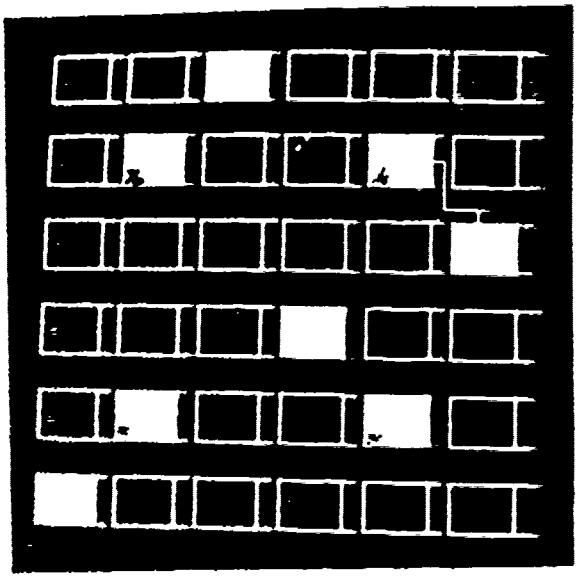


Figure 5-1: PARALLEL-MAP: Execution of the FOUR-CYCLE program. Active processors are displayed in inverse video. Active communications links are drawn as lines joing particular ports of the processor nodes. The processors hand-annotated with asterisks are the cycle centers. Each processor is at a different point in the cycle.

PARALLEL-LAMBDA:

```
;;; This creates a process at some other node in the network,
;;; returning an object which, when applied as a function to two
;;; arguments, evaluates a linear expression on those arguments.
;;;
(define (linear-evaluator a1 b1)
  (parallel-lambda t (x y) ';any-other ((a a1) (b b1))
      (+ (* a x) (* b y))))
```

MULTICAST-MAP-NO-REPLY (see Figure 5-2):

```
;;; This activates the processor at each location in SITES.
;;;
(define (activate-locations sites)
  (multicast-map-no-reply (lambda () *here*) sites))
```

MULTICAST-MAP (see Figure 5-3):

```
;;; This sends a message to each location in the list SITES, asking it
;;; to return its location.
;;;
(define (identify-yourself sites)
  (multicast-map (lambda () *here*) sites))
```

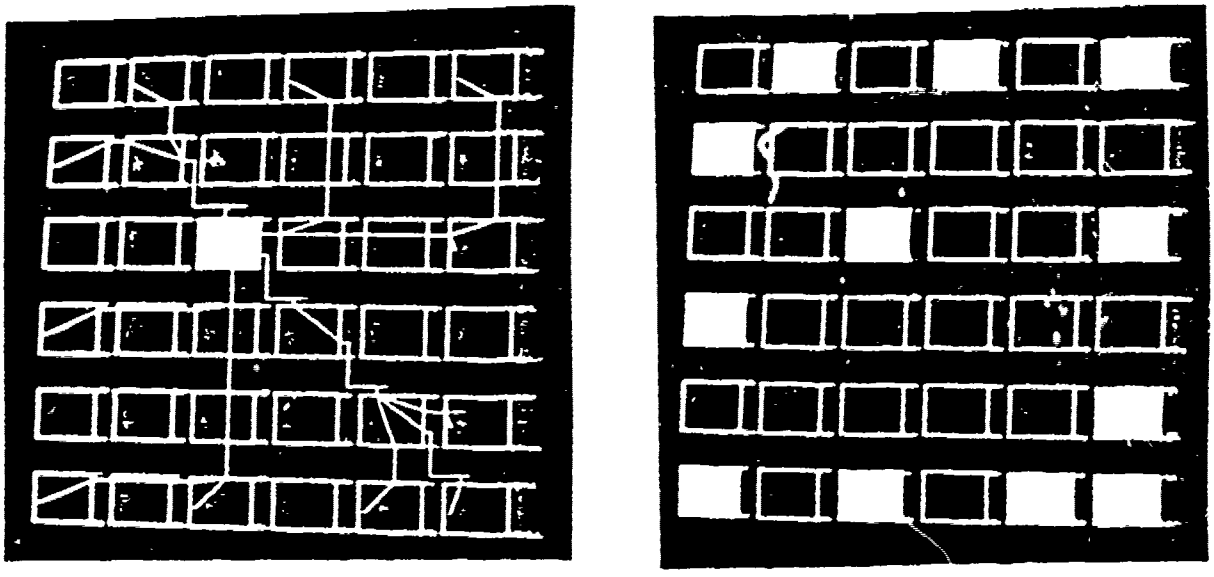


Figure 5-2: MULTICAST-MAP-NO-REPLY: Samples from the execution of the ACTIVATE-LOCATIONS program, showing how the multicast message is distributed and how the processors receiving the message are activated. Since no reply is required, the computation just dies out once the distributed programs are run.

PIPELINE:

```
;;; This sets up a pipeline across the bottom and up the right-hand
;;; side of the processor array.  This trivial pipeline simply adds
;;; 1 to the input value at each stage and passes the result on to
;;; the next stage.  It also prints out the result at each stage.
;;; using a printing mechanism "outside" the simulation.
;;;
(define (make-test-pipeline)
  (pipeline (s1 (x) '(1 6) ((a 1)) (print (+ a x)))
            (s2 (x) '(2 6) ((a 1)) (print (+ a x)))
            (s3 (x) '(3 6) ((a 1)) (print (+ a x)))
            (s4 (x) '(4 6) ((a 1)) (print (+ a x)))
            (s5 (x) '(5 6) ((a 1)) (print (+ a x)))
            (s6 (x) '(6 6) ((a 1)) (print (+ a x)))
            (s7 (x) '(6 5) ((a 1)) (print (+ a x)))
            (s8 (x) '(6 4) ((a 1)) (print (+ a x)))
            (s9 (x) '(6 3) ((a 1)) (print (+ a x)))
            (s10 (x) '(6 2) ((a 1)) (print (+ a x)))
            (s11 (x) '(6 1) ((a 1)) (print (+ a x)))))
```

6. Implementation

CAREL is implemented by a "semicircular"[1] interpreter, implemented in Zetalisp and drawing heavily on the CARE simulator. Some details of the implementation are provided in the full paper. These include the representation of CAREL datatypes, the use of a "global"

[1]Semicircular, not metacircular, because it is implemented in Lisp, but not in CAREL.

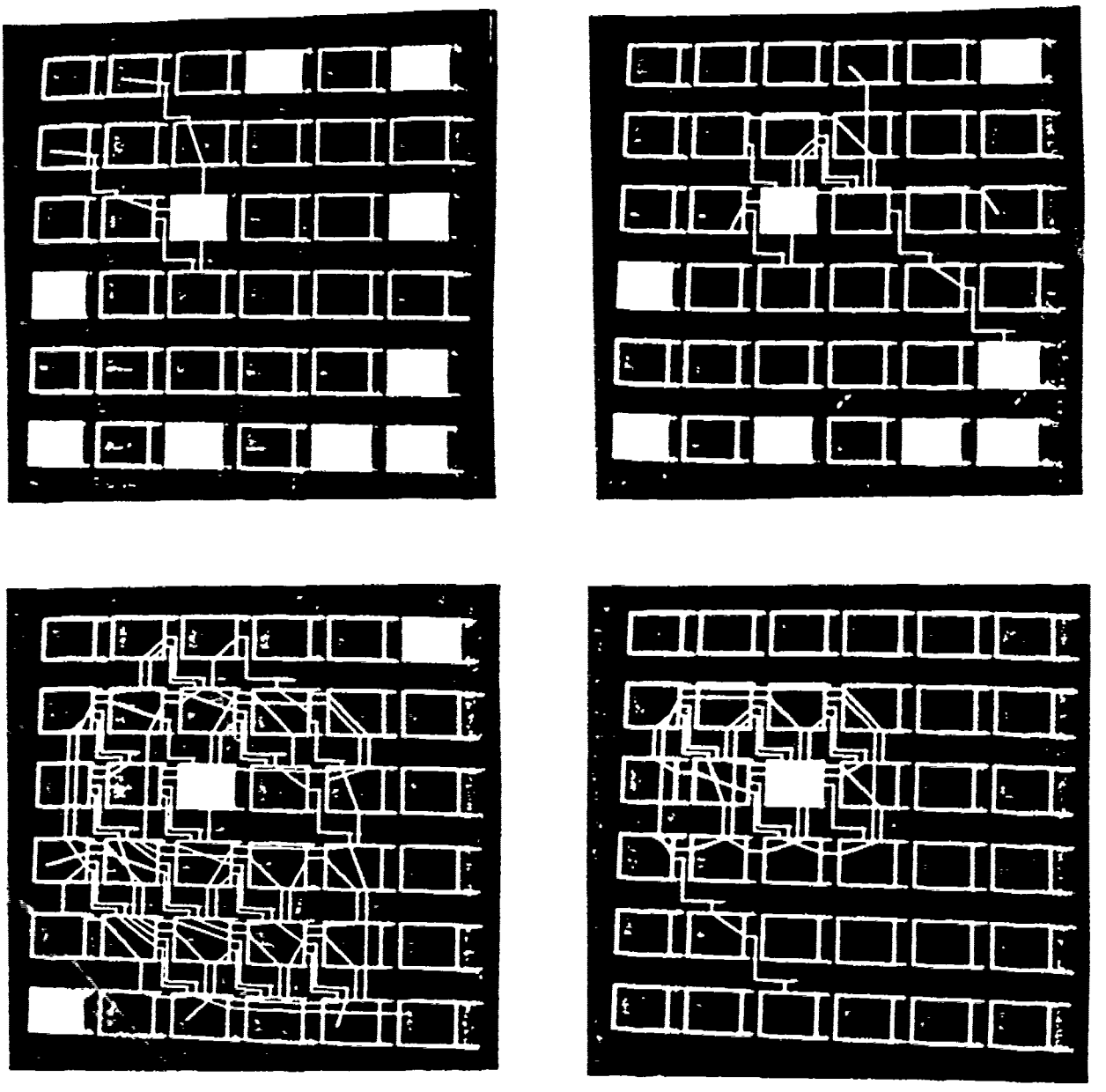


Figure 5-3: MULTICAST-MAP Samples from the execution of the IDENTIFY-YOURSELF program. The multicast method is distributed as in Figure 5-2, but in this example the processors must send a value back to the requesting process. The network becomes congested as all the processors respond then gradually returns to rest as the messages reach their destination. The notion of a network "heartbeat" is clearly demonstrated.

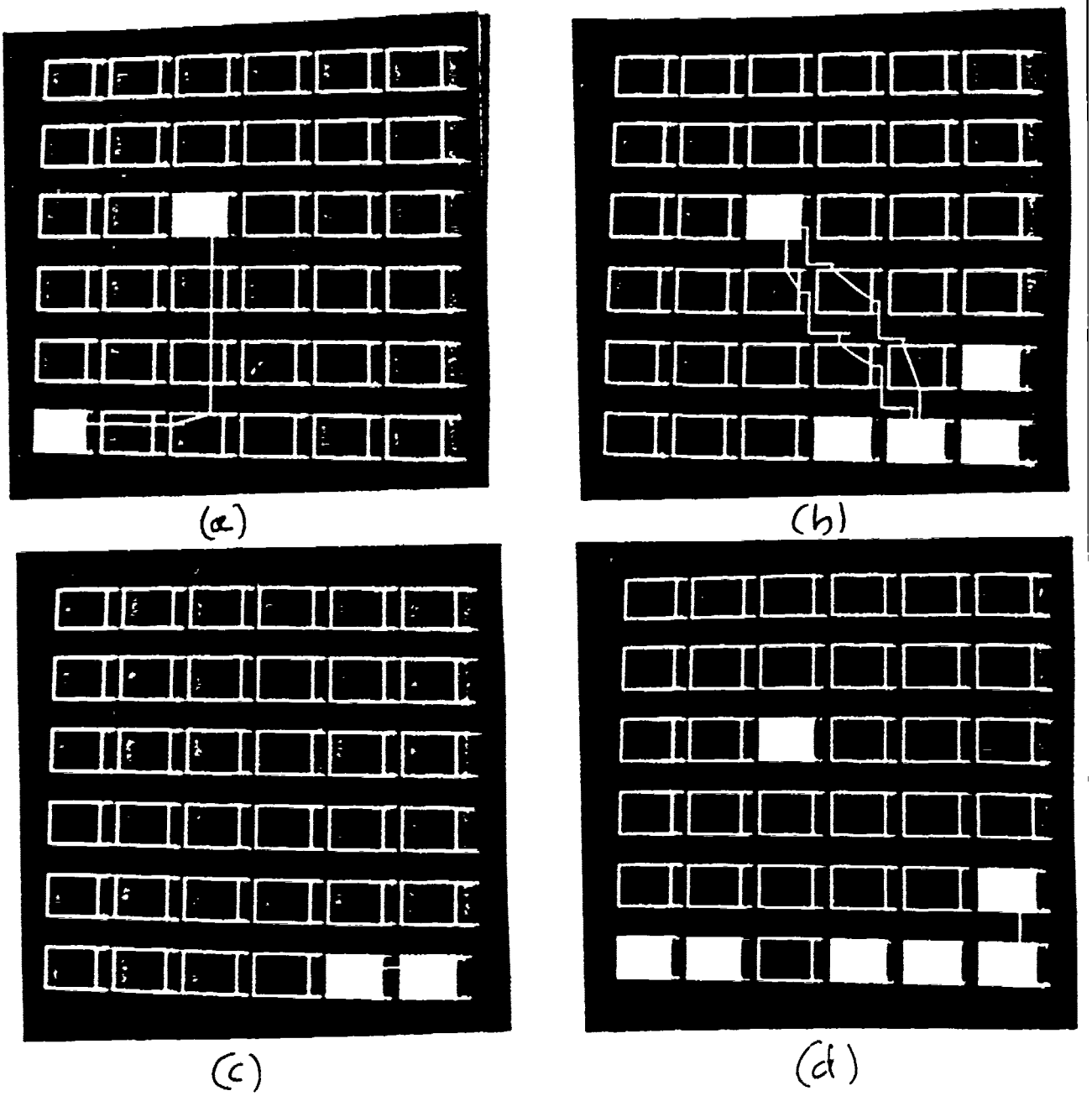


Figure 5-4: **PIPELINE**: Samples from the execution of programs constructing and using a CAREL software pipeline. The pipeline runs along the bottom and up the right side of the processor array. The pipeline is constructed in two passes. The first pass (a) establishes a process at each site and the second pass (b) links the processes together. The execution of the pipeline on a single argument (c) shows data flowing through the pipeline using only local communication. The last figure (d) shows multiple data items may flowing through the pipeline simultaneously, keeping multiple processors busy.

environment (full copies of which exist at each processor) and processor-local environments, and the interface to the CARE hardware simulator.

7. CAREL and Other Languages

CAREL was strongly influenced by three other languages: QLAMBDA [Gabriel and McCarthy 84], Par-Alfl [Hudak and Smith 86], and Actors [Agha 85]. QLAMBDA provided the idea of having two kinds of parallelism (which Filman and Friedman called parallelism by lexical elaboration and parallelism by explicit processes). CAREL addresses the question, "What would QLAMBDA look like on a distributed-memory multiprocessor?".

Par-Alfl provided the notion of a dynamic variable $self that a process could use, reflectively, to determine where it was executing. The part of CAREL that implements parallelism by lexical elaboration is very similar to Par-Alfl. CAREL adds the ability to deal with processes as first class objects.

Actors continues to serve as the "right thing" in the domain of languages for parallel symbolic computing. Calculating the difference between what CAREL can do and what Actors should do is always a valuable source of ideas for improvement. CAREL provides one particular set of primitives for describing both concurrency and locality. These primitives are powerful enough to implement a wide variety of interesting programs, but still provide less concurrency, less capability for managing synchronication, and less theoretical elegance than Actors. For example, CAREL enforces synchronization at the inputs and outputs of a function or closure: when APPLY is invoked, all the arguments must have been pre-evaluated, and multiple outputs are considered to be generated in a single list. In the Actor language SAL described by Agha, the inputs to an Actor may arrive at any time and in any order and outputs likewise may be generated asynchronously. Furthermore, Actors promise to make process management as invisible as memory management is in Lisp.

The plan for CAREL is to migrate it toward an Actor language. The CARE architecture is very close in spirit to the Actor approach, and would provide a nearly ideal environment for implementing Actors.

8. Acknowledgements

Implementation of CAREL was made possible by the existence of the CARE simulator, as implemented by Bruce Delagi and augmented by Eric Schoen. The author further wishes to acknowledge the intellectual support of the Stanford Advanced Architectures Project. Contributors to PARSYM, the netwide mailing list for parallel symbolic computing, have provided fruitful stimulation.

References

[Abelson and Sussman 85]
Harold Abelson and Gerald Jay Sussman with Julie Sussman.
Structure and Interpretation of Computer Programs.
MIT Press, Cambridge, Massachusetts, 1985.

[Agha 85] Gul A. Agha.
Actors: A Model of Concurrent Computation in Distributed Syster...
Technical Report, MIT AI Laboratory, March, 1985.

[Davis and Robison 85]
A. L. Davis and S. V. Robison.
The Architecture of the FAIM-1 Symbolic Multiprocessing System.
In *Proceedings of IJCAI-85.* 1985.

[Delagi 86] Bruce Delagi.
CARE User's Manual
Heuristic Programming Project, Stanford University, Stanford, Ca. 94305, 1986.

[Filman and Friedman 84]
R. E. Filman and D. P. Friedman.
Coordinated Computing: Tools and Techniques for Distributed Software.
McGraw-Hill, New York, 1984.

[Gabriel and McCarthy 84]
Richard P. Gabriel and John McCarthy.
Queue-based multiprocessing Lisp.
In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, August 1984.* 1984.

[Halstead 84] Robert H. Halstead.
Implementation of Multilisp: Lisp on a Multiprocessor.
In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, August 1984.* ACM, 1984.

[Hudak and Smith 86]
P. Hudak and L. Smith.
Para-functional programming: A paradigm for programming multiprocessor systems.
In *Proceedings of ACM Symposium on Principles of Programming Languages, January 1986.* ACM, 1986.

[Shapiro 84] E. Shapiro.
Systolic programming: A paradigm of parallel processing.
In *Proceedings of the International Conference on Fifth Generation Computer Systems.* 1984.

Knowledge Systems Laboratory
Report No. KSL 86-67

November 1987

LAMINA: CARE APPLICATIONS INTERFACE

by

Bruce A. Delagi, Nakul P. Saraiya, and Gregory T. Byrd

Knowledge Systems Laboratory
Computer Science Department
STANFORD UNIVERSITY
Stanford, California 94305

and

DIGITAL EQUIPMENT CORPORATION
Maynard, Massachusetts 01754

This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Greg Byrd was supported by an NSF Graduate Fellowship and by the Stanford University Department of Electrical Engineering.

ABSTRACT

LAMINA provides extensions to Lisp for studying expressed concurrency in functional programming, object oriented, and shared variable styles of computation. The implementation of the support for all three computational styles is based on the common notion of a *stream*, a datatype which can be used to express pipelined operations by representing the promise of a (potentially infinite) sequence of values. A pipelined algorithm to provide the sorted order of sequences of set elements is presented in the functional, object oriented, and shared variable programming styles for comparison.

In addition to demonstrating that a common set of primitives based on the notion of a stream is adequate for support of all three styles mentioned, LAMINA illustrates the means by which software pipelines may be managed and the means by which dynamic structure creation, relocation, and reclamation may be localized in a multiprocessor system.

Algorithms and applications written in LAMINA may be run on the SIMPLE/CARE simulation system in order to study their execution on alternative multiprocessor architectures. This has been done for two "expert system" applications and linear speedups over the range from one to eighty processors have been measured using LAMINA.

1 Streams, Values, and References

The SIMPLE/CARE multiprocessor simulation system [4] supports an applications programming interface, LAMINA, which currently is built upon Zetalisp [14]. LAMINA has been used as the basic programming language for two "expert system" application developments [2, 10] demonstrating significant speedup with increasing numbers of processors. LAMINA includes primitive mechanisms and language interface syntax for alternative approaches to the expression and management of concurrency and allows their relative performance to be measured on a common ground.

Functional, object oriented, and shared variable programming styles are all directly supported by LAMINA. The support provided for these styles is described in sections 2, 3, and 4 respectively. Section 5 describes some general utility functions. Primitives implementing the underlying mechanisms are described in an appendix. A second appendix lists the constructs of LAMINA and provides references into the body of the paper for details. The remainder of this section consists of background material describing how the values of one computation are passed to another and how the address space of an application is spread across the processors of a system in LAMINA.[1]

1.1 Futures and Streams

Futures [5, 6] and ***streams*** [8, 11] provide the common ground between functional, object oriented and shared variable programming in LAMINA. They are fundamental to the LAMINA functional and object oriented programming regimes for parallel programming and, since they are the only mutable items passed as references (rather than structure values) between potentially concurrent computations in LAMINA, they are also used to build the mechanisms for shared variable computation.

Futures and streams represent promises for values. We can arrange for promises for values, that is, their futures, to be used as placeholders in a computation while the values themselves are being *eagerly* [8] produced by concurrent evaluations for consumption as available. Extending this idea, we can define a ***stream*** as an abstract data type which is a placeholder representing a sequence of eagerly produced but potentially unavailable values.

Some operators do not require the actual values promised by a stream or future in order to perform their work. For example, a constructor may create data structures that include streams as structure elements. The creation can be accomplished without accessing any of the promised values that the streams represent; referencing streams as placeholders is sufficient. Further, streams, as sequences of potentially unavailable but eagerly produced values, can be used to build pipelines of computation connecting the producers and consumers of such values.

Streams may be arguments to or the results of function application. In LAMINA, streams are a primitive data type developed for use in an object oriented programming style and futures are a specialization of streams that represent only a single (potentially unavailable) value as required for the functional programming style. Streams and futures are always passed as references. In the remainder of the paper, the term *stream* or *future* is equivalent (respectively) to a *reference* to a stream or a future.

1.2 Processor Address Spaces and Multilevel Allocation

In LAMINA, structures of arbitrary complexity can be supplied as a value of a stream or future either local or remote to the processor address space in which the structure was generated. Internal pointer references within copies of such structures are adjusted (for address relocation) as the copies pass between the originating processor address space and the processor address space of the stream that represents the promise for the values so supplied. External pointer

[1] Footnotes in the paper generally deal with details, conventions, or implemenation issues that can be skipped on first reading.

references included in structures passed between spaces are restricted in LAMINA to locations in global *dynamic* or *static* address spaces as shown in figure 1. Statically allocated structures are not relocatable or reclaimable and may be regarded as cacheable and immutable. Thus, they may be globally referenced without a need for access coordination.

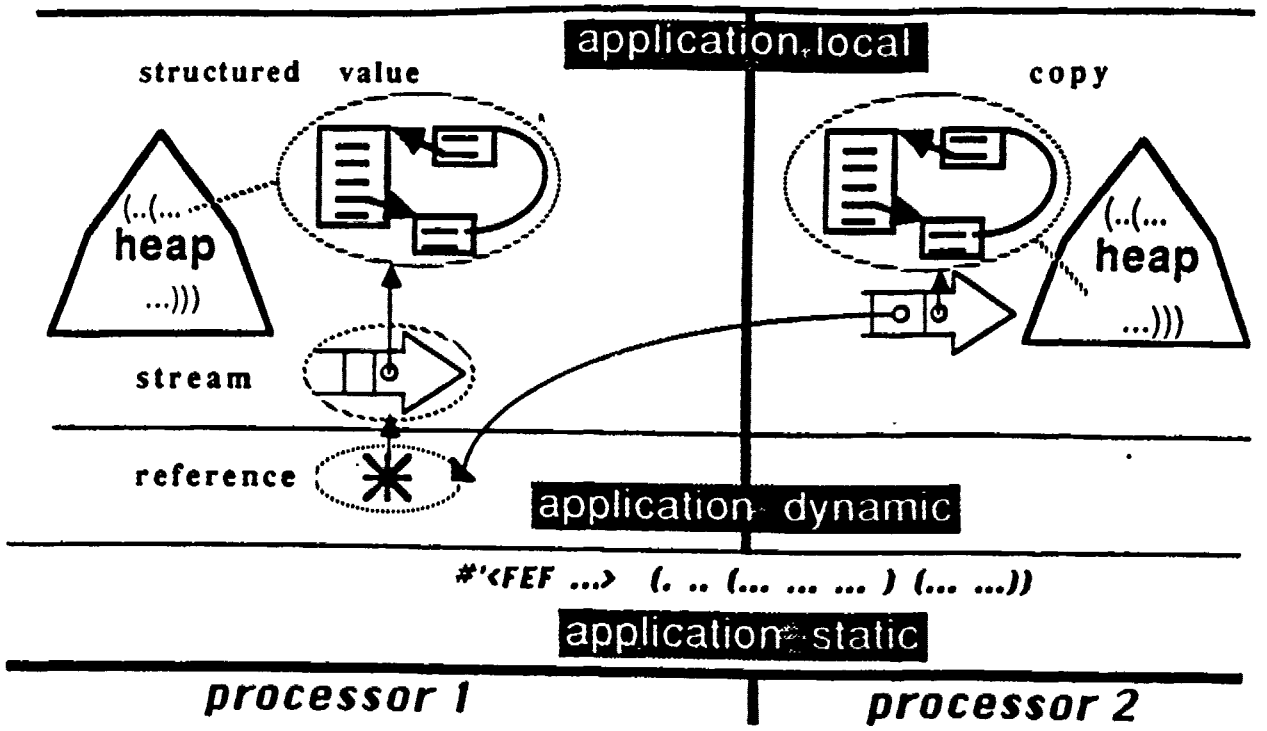


Figure 1: LOCAL, DYNAMIC, & STATIC ADDRESSES

When values are passed between processor address spaces the structure representing the value, that is, the *structure value*, is recursively copied until a data structure is produced which has the same form and internal relationships as the original value but which holds only: *static references* (to code bodies and other structures in static space), *dynamic references* (to streams or other structures) in dynamic space, *internal references* (to subcomponents of the structure value), and *self-referentials* (for example, numbers and characters).[2] Copying of a structure value might be done asynchronously with evaluation of the user application, so if changes are to be made in the structures encompassed by a structure passed between address spaces, independent copies of such structures should be formed.

An example of values and references passed between processor address spaces is shown in figure 1. One of the values of the indicated stream in the application's processor 2 *local* address space is a copy of the structure value in the application's processor 1 *local* address space. Both structure values are heap allocated from independently managed heaps in separate local spaces. Allocation, relocation, and reclamation for each given heap may be done asynchronously based on just the information in the associated processor address space. The other value shown for the indicated stream in figure 1 is a *reference* (in this case, to the original structure value) allocated in the application's *dynamic* space. Because the reference and its associated structure value are allocated within a single processor, relocation of the locally allocated structure value can be done locally and asynchronously. Relocation of the reference, however, must be globally coordinated. Statically allocated structures are not relocated or reclaimed.

[2] As a current implementation restriction, lexical closures [12] passed between processor address spaces may only be made over free variables whose values are references or self-referentials items and not structures that contain them.

References to streams are allocated in dynamic space and streams are accessed by reference. A stream reference, therefore, may only be relocated (for example as required by a compacting garbage collector) through globally synchronized operations affecting all computations that could access that stream. This global synchronization can be expensive and involve subtle low level implementation considerations. Expectations about the expenses involved in correct global syncronization[3] led the design of LAMINA to a multi-level allocation scheme described below.

The cheapest approach to allocation (and deallocation) of memory for dynamically created structures is *stack-based* (and local). However, the benefits of stack-based operation come at the cost of a prescribed order of deallocation. Additionally (at least for the commonly used memory management enforced stack limit schemes), stack-based operation entails a minimum storage commitment that is significantly larger than the rest of the execution environment for each highly concurrent, small granularity evaluation expected in LAMINA programs. Stack based allocation is used in LAMINA whenever references to structures with dynamic extent [13] are known to be entirely within a given sequential computation.

The next cheapest approach, for references that are local with indefinite extent [13], is heap based allocation in *local* space. Since such references are confined to a single processor address space, they may be relocated asynchronously with operations on other processors and memories or in the network connecting the components of the multiprocessor system.

Finally, as the most expensive approach, global references may be made to dynamically allocated references (which must be relocated under a global synchronization scheme). Allocation in *dynamic* space is done independently by each processor and each allocation is distinct. Operations involving dynamically allocated references are handled by the processor (or memory controller) associated with the reference. The referents for such references are mutable and may be viewed as uncacheable.

References to locally allocated structures can also be passed between processor address spaces by encapsulating them in dynamically referenced structures, that is, streams. By this indirection, pointers to selected locally allocated structures are held locally (and may readily be relocated) but a means is provided to reference them in other processor address spaces.

The multi-level allocation scheme just described creates references passed between processor address spaces (with the attendant synchronization expenses) only as necessary. The remainder of this section describes the syntax for creating and accessing such references.

1.3 Reference Creator and Accessor Functions

When a locally allocated data structure needs to be passed between potentially concurrent computations as a reference rather than as (a copy of) its value, the form (`reference` *item*) returns a reference for the value of the item.

The *site* of a reference, that is, the CARE processor (or memory controller) on which it was created, may be determined by executing (`reference-site` *reference*). The value returned by calls to this function is a *site refrence* that may be used to specify sites as required as parameters of other LAMINA functions.

Finally, references can be tested to determine whether they refer to the same item by the function `reference-eq`, a function that accepts two references as arguments and returns a non-nil value if they refer to the same item.

[3] For example, in a shared memory system with asynchronous writes to memory, a request to change the contents of a location in dynamic space so that it points to a stream in a given semispace of a compacting garbage collector may have been in transit to a *memory controller* when evacuation of that semispace was requested. The evacuation must be delayed somehow until all such requests either in transit or queued anywhere in the system have been processed. Shared memory systems with synchronous writes delay *all* processor operations on shared variables until the *memory* request can first traverse the network between processors and memories (or other caches), then be queued and serviced in the memory (or other cache) controllers, and finally traverse the network back to the processor.

2 Functional Programming

Perhaps the style of computation most readily treated as concurrent is that of functional programming. LAMINA supports concurrent programming using this style by providing means (1) to spawn computations that will provide values to futures and (2) to accept such values in a computation -- scheduling the computation when they are available. The constructs defining the LAMINA interface for functional programming are:

- (`future` *form*) spawns execution of a *lexical closure*, that is, a procedure body to execute a given form together with an environment (determined by the rules of lexical scoping) in which to do the execution [13]. This closure is executed (eagerly) on a randomly selected site. A future which will contain the value of the computation when it is available is immediately returned.
- (`with-values` *future-bindings forms*) spawns an evaluation on the local site to execute the closure corresponding to the *forms*. The evaluation is done within an environment that includes bindings for given variables to the values available for the indicated futures. The evaluation is deferred until all of the indicated futures have values that are not themselves futures. The immediate result of executing a `with-values` form is a future whose value will be supplied by the deferred evaluation.

Each element of a *future-bindings* list is itself a list: (*binding-pattern future-specifier*). If evaluation of a future specifier in a `with-values` construct produces a value other than a future, the future specifier is coerced to be a future holding that value. After all specified futures have values (which are not themselves futures), the values of each of the futures are *destructured* [13], that is, the values are treated as list structures and the elements of these list structures are used to bind corresponding variables in a binding pattern of arbitrary depth. These bindings will be included in the environment in which the spawned computation is executed. Only `with-values` can be used in LAMINA to reduce futures to values. Values of futures are never taken as an ancillary consequence of any other operation.

The results of the evaluation spawned by `with-values` are returned as a future which will receive the value of the spawned computation. The spawned evaluation that is created by a `with-values` construct is treated as the continuation [12] of the computation in which it is found and, as such, captures all stack allocated values required to execute that computation. Thus, each spawned computation may be viewed as running to completion; its continuation, if any, is an independent spawned computation.

Because all spawned computations run to completion (unless they are preempted by system level operations), the stack of the executing processor is (generally) left clear and any space allocated for it may be reused by the next computation on that processor. By this means, the advantages of stack-based operation are retained without incurring the space penalty discussed in section 1.2. The costs of heap allocation are incurred only as needed.

To illustrate the use of the LAMINA functional programming interface, the implementation of a (quicksorting) algorithm to associate ordering information with the numerical values of the elements of sets supplied as input is shown in figure 2. The serial and parallel implementations may be compared by contrasting the definitions of the functions `order0` and `order1`.

The input to the ordering functions is sets of numbers to be ordered. Elements of a set are the sequential elements of a list before a separator token (which is `nil`). The sets (including their separator tokens) are concatenated to form the input list. The output is a list with each ordered set represented by successive elements of a list and separated from other ordered sets by `nil` tokens. The sets follow each other in the output in the same order in which they appeared in the input. For example, the input list (`7 9 4 nil 5 3 8 nil`) would result in the output (`4 7 9 nil 3 5 8 nil`). Thus the information concerning the ordering of the elements of a set and the identity of that set is implicit in the output.

In `order0` and `order1`, the result of ordering `nil` is `nil`. If the input list is not `nil`, the

```
(DEFUN ORDER0 (input-list)
  "Serial quicksort to order elements of input sets"
  (if (null input-list) nil
      (let ((pivot (car input-list)))
        (if (null pivot) '(nil . ,(order0 (cdr input-list)))[4]
            (destructuring-bind (smaller larger rest)
                                (part1 pivot (cdr input-list))
              (let ((ordered-smaller (order0 smaller))
                    (ordered-larger (order0 larger))
                    (ordered-rest (order0 rest)))
                '(,@ordered-smaller ,pivot ,@ordered-larger
                  . ,ordered-rest)))))))

(DEFUN ORDER1 (input)
  "Without pipelining: recursively spawn ordering partitioned input sets"
  (with-values ((input-list input))
    (if (null input-list) nil
      (let ((pivot (car input-list)))
        (if (null pivot)
            (with-values ((rest (order1 (cdr input-list))))
              '(nil . ,rest))
            (destructuring-bind (smaller larger rest)
                                (part1 pivot (cdr input-list))
              (with-values ((ordered-smaller (future (order1 smaller)))
                            (ordered-larger (future (order1 larger)))
                            (ordered-rest (future (order1 rest))))
                '(,@ordered-smaller ,pivot ,@ordered-larger
                  . ,ordered-rest))))))))

(DEFUN PART1 (pivot input-list)
  "Serial: add elements from input list sets into one collection or other"
  (let ((input (car input-list)))
        (if (null input) '(nil nil ,input-list)
            (destructuring-bind (smaller-part larger-part rest)
                                (part1 pivot (cdr input-list))
              (if (> input pivot)
                  '(,smaller-part (,input . ,larger-part) ,rest)
                  '((,input . ,smaller-part) ,larger-part ,rest))))))
```

Figure 2: FUNCTIONAL ORDERING

first element of that list is used as a `pivot`. If that element is `nil`, it is a separator token. The result then is the separator followed by the result of ordering the rest of the list. If the pivot element is not `nil`, it is assumed to be a number that is used by `part1`, a serial partitioning function which returns a list of three results: the (unordered) elements of the current set smaller than the pivot, the (unordered) elements of the current set larger or equal to the pivot, and the remaining elements of the input.

The function `order1` spawns executions to apply itself to each of the three sublists returned by `part1` to order them. It then waits for the results. When these are available, it appends the ordered sublist of elements that were smaller than the pivot to the list formed by the pivot, the ordered sublist of elements that were not smaller than the pivot, and the result of ordering the rest of the sets in the input.

The operation of `order1` is characterized by much waiting for the results of spawned

[4] Due to printing limitations, the backquote character will appear as '. Inclusion of a comma in the form introduced by a backquote will disambiguate the quoting character.

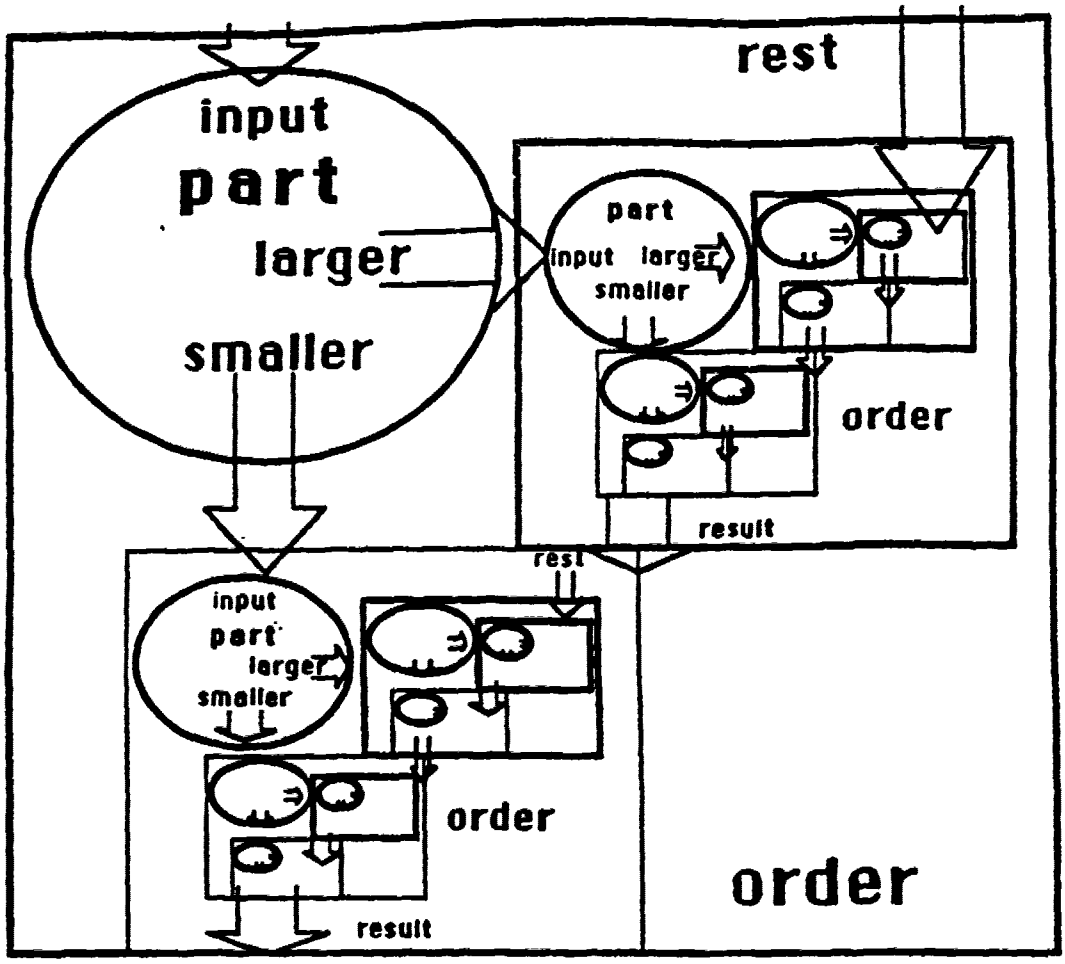


Figure 3: ORDERING PIPELINE

computations. The pattern of execution is to spawn a set of computations -- using `future` constructs -- and immediately wait for all their values to be produced -- using `with-values` constructs. This waiting represents serialization due to data dependencies and can significantly limit the concurrency of an algorithm. If, instead, computations can be handed just what they each require to get started (with promises for the rest), they can be pipelined as computation assembly lines, each station operating on a piece of the input from upstream producers and delivering a piece of the output to downstream consumers.

A schematic view of a pipelined ordering algorithm is shown in figure 3 while the code is shown in figure 4. The schematic is a recursive drawing terminating in a number of ordering computations -- one leaf for each element and separator token in the sets of elements to be ordered. Each non-leaf node of the ordering tree partitions its input by sending each input element it receives (from its upstream parent) to one of its two downstream children. The smaller child was created such that its result is used as the result that the parent was asked to produce and the rest of its input is the result of the larger child. The larger child was created so that if it is a leaf (that is, if it has nothing to order), its result will be the rest of the items given to the parent. The rest of the items seen by the largest descendent of the smaller child is the result produced by the smallest descendent of the larger child. Thus, using an approach similar to the use of *difference-lists* in logic programming [11], the results of the leaf elements are tied together to produce the result of the ordering tree.

The first input a child receives will establish the pivot for partitioning unless it is the separator token, `nil`. If it is `nil` and there is more input, the child returns `nil` as the first part of the result together with a promise for ordering the rest of its input followed by those

```
(DEFUN ORDER2 (input-future &optional rest-pair)
  "Future pipeline: rest and input pair (or its future) => ordered pair"
  (with-values (((pivot . rest-input) input-future)) ; Coerce value
    (if pivot    ; Spawn partitioning and get promises for first elements
        (with-values (((smaller-future larger-future)
                       (future (part2 pivot rest-input))))
          (let* ((ordered-larger-future ; Spawn order larger
                  (future (order2 larger-future rest-pair)))
                 (ordered-larger-pair
                  '(,pivot . ,ordered-larger-future)))
            ;; Continue ordering smaller
            (order2 smaller-future ordered-larger-pair)))
        (if (null rest-input) rest-pair
            '(nil . ,(future (order2 rest-input rest-pair)))))))))

(DEFUN PART2 (pivot input-future)
  "Produces (<future> <pair>) or (<pair> <future>) for (<smaller> <larger>)"
  (with-values ((input-pair input-future))          ; Coerce value
    (if input-pair ; Destructure pair as (value . future)
        (destructuring-bind (input-value . rest) input-pair
          (if (null input-value) '(nil  (nil . ,rest))
              ;; Spawn continuation of this partitioning
              (let ((future-part (future (part2 pivot rest))))
                ;; and get futures for destructured value of continuation
                (let ((smaller-future
                       (with-values
                         ((value future-part)) (first value)))
                      (larger-future
                       (with-values
                         ((value future-part)) (second value))))
                  ;; Return list: (<future> <pair>) or (<pair> <future>)
                  (if (> input-value pivot)
                      '(,smaller-future
                        (,input-value . ,larger-future))
                      '((,input-value . ,smaller-future)
                        ,larger-future)))))))))
```

Figure 4: PIPELINED FUNCTIONAL ORDERING

values larger than anything in that input. If there is no more input, it just returns promises for the results of its larger relatives, that is, the `rest-pair`.

The receipt of a separator token while partitioning indicates that all the elements of a set to be ordered have been received. A terminator, `nil`, is passed to the smaller child and a separator followed by the rest of the unordered input (if any) is passed to the larger child.

The code for this example is written assuming that each stream can only hold one value, that is, streams are restricted to be simple futures. In the example, sequences of values are represented by pairs consisting of a value and a future for the rest of the sequence. The value of the future, when available, is a pair which itself consists of a value for the next element in the sequence and a future for the rest of the sequence. The consequence of this approach is that many short lived dynamic references are created (so that each element of the sequence has an independent reference) and then abandoned. Reclaiming the space allocated for them requires global synchronization as discussed in section 1.2.

Relaxation of the single value assumption for structures representing unavailable values -- as well as extension of LAMINA to an object-oriented programming style -- is discussed in the following section.

3 Object Oriented Programming

In LAMINA's object oriented programming interface, an object encapsulates related state variables and is referenced throughout an application by that object's `Self-Stream`, a stream (whose reference is in *dynamic* space) which is one of the object's state variables. Objects are allocated in *local* space as described in section 1.2. To perform operations on an object, potentially involving and modifying its state variables, a ***task request posting*** consisting of a ***task selector*** and associated parametric values for the operation is *sent to*, that is, provided as one of the values of the self-stream for that object. Each of the task request postings that provide the values for the self-stream of a object is taken in turn from that stream and serviced by that object.

Task request postings are serviced atomically in the context of an object. Executions specified by such request postings are done without visible partition with respect to other operations on that object: operations on any given object will not be interleaved. Each operation is thus defined to be ***independently atomic.***

All the operations on an object done as specified by the requests are taken in turn from the object's self-stream. Each operation runs to completion. If an operation on an object is preempted (due, for example, to page faulting, schedule quanta lapse, or error condition), no other operation on that object will be started before the preempted operation is completed. However, operations on other objects may proceed normally. A stack is maintained for each preempted operation.

3.1 Sending a Task Request

Sending a task request in LAMINA is non-blocking and thus pipelined operations on objects are directly accomodated. The information required to accomplish a task is either passed with the request or is included in the state variables of the object. In an object oriented programming style, state is localized in objects and is not referenced otherwise. Arbitrarily structured values, however, may be sent in task request postings between `lamina` objects as (copied) values rather than as references. Additionally, as is common in object oriented programming languages, *references* may be sent in task request postings as well.

The construct for asynchronously sending a task request posting to a target self-stream of an object resembles the Zetalisp (synchronous) `send` construction:

(`sending` *self-streams task-selector value lamina-keyword ...*)

Multiple targets for a posting may be specified as a target list and LAMINA keywords (as listed in figure 5) can be used to provide additional control or debugging information. For example, the task request may be sent with a *tag* field that can be used as a descriptive auxiliary value for debugging purposes.

The value immediately returned by `sending` is the list of *clients* supplied following the LAMINA keyword "`for`" (or `:for-effect` if no clients are specified). As a convention, the *clients* may expect to receive consequent task requests later in the computation.

3.2 Creating a New Stream, Ordered Stream, or Sequenced Stream

The streams that pass values between objects are created by the supplied function `new-stream`. Streams may be tagged for debugging purposes by including a tag as the optional first argument of `new-stream` as in (`new-stream` *tag*). The default argument, `nil`, will cause a stream to inherit a tag identifying the execution in which the call to `new-stream` appears.

The `new-stream` function returns a reference for a stream created on the executing site. Often, the reference for a stream (for example, the self-stream of an object) is passed by a procedure as a way of telling some other procedure how the executing (or some other) procedure expects to receive values to use or tasks to accomplish.

A stream may be thought of as an ordered queue of postings. Information can be included in

TO, ON *targets*	A target stream (or site) or list of targets streams (or sites) for the indicated LAMINA operation. If no site is provided and one is needed, an unspecified site is chosen. Some LAMINA operations expect site targets rather than stream targets. These are documented as they are introduced. The choice between the alternative keywords shown is purely stylistic.
FOR *clients*	A stream or list of streams acting as the continuation of the computation that will be triggered by the LAMINA operation.
AS *tag*	Arbitrary data for debugging. Defaults to the tag of the sending execution.
BY *order-key*	A number which may be used to order information in target streams.
AFTER *delay*	Positive number indicating the number of milliseconds that the operation will be delayed before being attempted.
WITH *properties*	Arbitrary data intended for user extensions of the posting protocol.

Figure 5: LAMINA KEYWORD VALUES

postings to allow them to be ordered in streams by specifying a value following the keyword "by" in the call creating the posting. A stream ordered by increasing numeric keys can be created by the function, `ordered-stream`. The function takes an optional argument for a tag: (`ordered-stream` *tag*).

As an optimization to simplify programming and to reduce scheduling overhead (by deferring executions involving out of order task invocations), a stream can be created that only presents queued postings that have order keys less than or equal to the next expected order key. This key is greater than or equal to zero and is one more than the highest order key of any previously presented postings. Thus, in the simplest case, the presented postings will have order keys that are in the sequence of the integers beginning with zero. The function, `sequenced-stream`, that creates such streams also takes an optional argument for a *tag*.

Streams that have at most one value may be created by the function `new-future`. This function too takes an optional argument for a *tag*.

3.3 Defining Objects

LAMINA object types are built upon the base *flavor* [9], `lamina`, which defines the instance variable, `Self-Stream`. The default specification is for a first-in-first-out self-stream. Flavors intended to be mixed in to `lamina`, the "mixins" `ordered-self-stream` and `sequenced-self-stream`, are provided to override this default. As an example similar to the one discussed in section 2, a LAMINA object to associate ordering information with the numerical values of the elements of sets might be defined as shown in figure 6. In the example, the state variables of an ORDER3 *ordering object* are all named, the default initializations specified, and any state variables to be initialized by a creator are identified.

3.4 Triggers

Task request postings specify a task-selector, a value, and the information associated with the keywords in the posting that originated the request. The value and other information in the posting is formatted as a list: (*value clients key tag origin properties*). This list is destructured for execution according to the *trigger-pattern* specified in the trigger definition. Posting elements that are to be ignored need not be specified and an arbitrary degree of destructuring can be specified by the trigger pattern.

```
(DEFFLAVOR ORDER3 ((Controls[6] (ncons :controls))
                   (Smaller-Child) (Larger-Child) Id Result-Stream)
 (lamina)
 (:initable-instance-variables Id Result-Stream)) ; This must be specified

(DEFTRIGGER (ORDER3 :ELEMENT) (input)
  "Set pivot or partition by established pivot.  Check for completed set"
  (destructuring-bind (value set-id) input
    (let* ((control (send self :control set-id))
           (pivot (control-pivot control)))
      (if (null pivot) (setf (control-pivot control) value)
          (if (>= value pivot)
              (sending Larger-Child :element input)
              (sending Smaller-Child :element input)
              (incf (control-smaller control)))) ; Count smaller in set
      (send self :completed? control set-id))))

(DEFTRIGGER (ORDER3 :END) ((base set-id expected))
  "Note base and send :end to children if complete"
  (let ((control (send self :control set-id)))
    (setf (control-expected control) (1+ expected))
    (setf (control-base control) base)
    (send self :completed? control set-id)))

(DEFMETHOD (ORDER3 :CONTROL) (set-id)
  "Get or create control for input and make descendants if none ever made"
  (when (null Smaller-Child)
    (setq Smaller-Child (new-stream) Larger-Child (new-stream))
    (creating 'Order3 '(:Self-Stream ,Smaller-Child :Id (< ,Self-Stream)
                        :Result-Stream ,Result-Stream))
    (creating 'Order3 '(:Self-Stream ,Larger-Child :Id (>= ,Self-Stream)
                        :Result-Stream ,Result-Stream)))
  (or (get Controls set-id) (putprop Controls (make-control) set-id)))

(DEFMETHOD (ORDER3 :COMPLETED?) (control set-id)
  "Count received in set against expected and finish off set if complete"
  (let ((expected (control-expected control)))
    (when (eql expected (incf (control-count control)))
      (let ((pivot (control-pivot control))
            (base (control-base control))
            (smaller (control-smaller control)))
        (let ((pivot-order (+ base smaller))
              (larger (- expected smaller 1)))
          (sending Result-Stream :element '(,pivot ,set-id ,pivot-order))
          (let ((new-base (1+ pivot-order)))
            (if (plusp smaller)
                (sending Smaller-Child :end '(,base ,set-id ,smaller)))
            (if (plusp larger)
                (sending Larger-Child :end '(,new-base ,set-id ,larger))))
          (remprop Controls set-id))))))

(DEFSTRUCT (CONTROL :conc-name :named)
   ((pivot nil) (base nil) (expected nil) (count 0) (smaller 0)))
```

Figure 6: OBJECT ORDERING

[5] As a convention, capitalized names are understood to refer to the state variables of an object.

The syntactic form for trigger definition is modeled after the Zetalisp DEFMETHOD form:

(DEFTRIGGER (*object-type trigger*) *trigger-pattern*
documentation-string . *trigger-body*)

Example trigger definitions for an ordering object are shown in figure 6. Iteration and assignment replace the recursion and binding used for the functional programming ordering example shown in figure 4. Sequences of values on streams are represented by long lived streams that couple producing and consuming ordering objects.

In the example, each :element message manipulated by the ordering routine indicates the value of the element to be ordered and the set in which that element appears. The output :element messages include this information together with the calculated order of the element in the indicated set. An :end message may be generated either by the root calculation requesting a set be ordered or by intermediate ordering objects serving that calculation. Each such message includes a set identifier, the number of elements the receiver should expect for that set, and the (base) order of the smallest element to be expected. The ORDER3 objects keep track of this (and other) information for each set they are dealing with in a (disembodied property) list of control records. The set of an input is used to retrieve the appropriate control record from among those in use by the object.

If there is no pivot yet received to use in partitioning the set, the ordering object saves the input value as the pivot for the set. Otherwise, the :element trigger method passes the input element to either its larger or smaller child and counts the number of elements sent to the smaller child. If all the expected inputs for a set have been received, an :element message including the value, the set, and the order of the value in the set will be sent to the result stream. An :end message will be sent to any children that have been sent elements of the set to order.

3.5 Creating LAMINA Objects

The form (creating *type initializations* for *client-streams* on *site* ...) stipulates the creation of a object on the indicated site (or on a randomly selected site if none is indicated). When the creation has been accomplished, the client streams will receive a posting whose value is the self-stream of the created object.

The *initializations* are formed as a list alternating keywords (corresponding to the state variable names for the object being created) with their initial values. These values are computed in the context of the object requesting creation. As an example, creating forms are included in the ORDER3 :control method definition shown in figure 6.

For convenience, a function, create-self-stream, is provided to create a stream which is either an ordered stream, a sequenced stream, or a FIFO stream as appropriate for the self-stream of the lamina object type specified by its argument.

An example of a trigger definition to create three intercommunicating objects is shown in figure 7. In the example, three objects each with state variables referencing the self-stream of each of its siblings are created together. State variables of each object representing an id for the triplet and the object that requested the creation are initialized as well.

3.6 Implicit Continuations

For LAMINA objects, continuations of a computation are often some explicit trigger method of some explicit object. There are cases, however, in which it is inconvenient to create an explicit name for a continuation. As a syntactic construct, execution of a continuation of a computation can be specified to occur in the context of an executing object (as defined by its set of state variables and the environment of the continuation) each time that postings have been received on some given streams. The execution spawning the continuation is finished normally and then the next operation to be done on the object is taken from its self-stream without delay. Thus LAMINA objects can be viewed as *monitors* [1] (because the independently

```
(DEFTRIGGER (TRIPLICATOR :ABC-TRIPLET) (id client)
  "Expect created object to send notice of its creation"
  (let ((a-stream (create-self-stream 'a))
        (b-stream (create-self-stream 'b))
        (c-stream (create-self-stream 'c)))
    (creating 'a (list :Self-Stream a-stream
                       :B b-stream :C c-stream :Id id :Parent client))
    (creating 'b (list :Self-Stream b-stream
                       :A a-stream :C c-stream :Id id :Parent client))
    (creating 'c (list :Self-Stream c-stream
                       :A a-stream :B b-stream :Id id :Parent client))))
```

Figure 7: COUPLED OBJECT CREATION

atomic operations on objects give the required mutual exclusion) but operations on them are unnested. This is done to facilitate pipelined operation: task request postings queued for operation on an object are not deferred for a pending continuation.

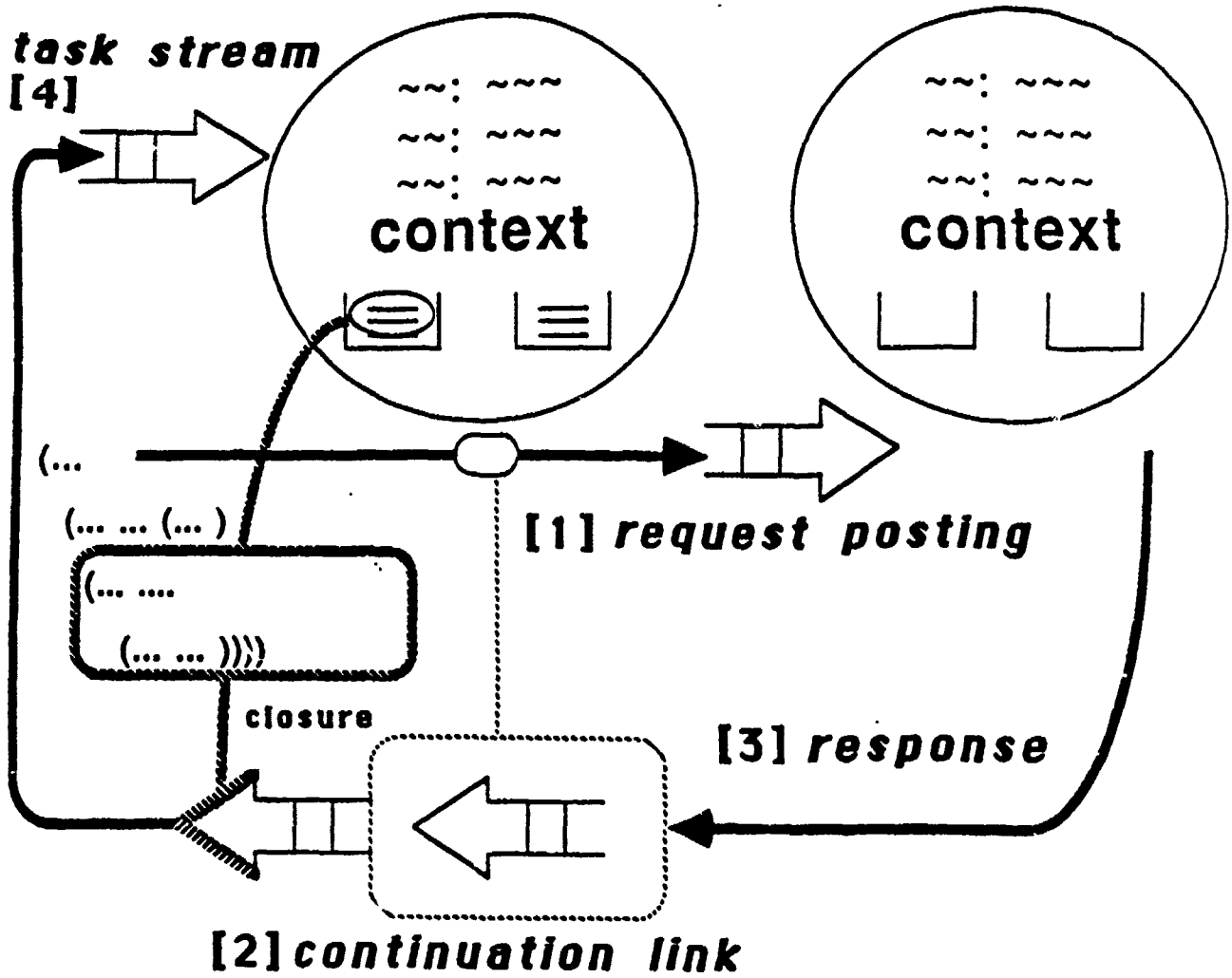
The construct (with-postings *stream-bindings form*) creates an implicit continuation in the context of an object. The *stream-bindings* is a list each element of which is of the form (*binding-pattern stream*). Each of the postings on the indicated streams (including the posting clients, tag, key, origin, and properties) will be destructured and bound to a corresponding variable (identifier) according to the associated *binding-pattern*. These variables and associated values are also part of the execution environment of the continuation.

```
(DEFTRIGGER (DISTRIBUTER :MAKE-A   SERVERS) ((count input-stream))
"Round robin distribution of input requests to created triplets of servers"
  (let ((a=> (creating 'a nil for (new stream)
                       on (loop repeat count collect (random-site))))
        (b=> (creating 'b nil for (new-stream)
                       on (loop repeat count collect (random-site))))
        (c=> (creating 'c nil for (new-stream)
                       on (loop repeat count collect (random-site))))
        (servers (ncons nil)))
    (with-postings ((a a=>) (b b=>) (c c=>))
      (if servers (rplacd servers (cons (list a b c) (cdr servers)))
          (setq servers (circular-list (list a b c)))
          (with-postings ((request input-stream))
            (sending (pop servers) :request request as Self-Stream))))))
```

Figure 8: WITH-POSTINGS

As an example of the use of with-postings, we can consider the example shown in figure 8. It uses nested with-postings constructs to create continuation closures that create and collect triples of lamina nodes and then distribute requests on an input stream to the collected triples in a round robin fashion. Note that instance variables may be accessed by the continuations.

The implicit continuation will be executed atomically with respect to any other operations on the indicated object and in the context of its state variables and the lexical environment in which the form appears. A schematic of the mechanism supporting implict continuations in objects is shown in figure 9.



[1] References for streams on which responses are expected are sent in (task request) postings to other objects as places to supply response postings. [2] Intermediate variables (that is, the environment) and a pointer to a block of code required to execute the form wrapped in a `with-postings` *construct are captured in a continuation closure, attached to a stream, and linked to the stream(s) on which responses are expected. [3] When all required postings become available on these streams, [4] the response postings together with the closure are sent to the self-stream of the object that generated the closure.*

The closure is executed (in its turn) atomically within the context of the object and lexical environment of the form. Variable bindings are made as specified to the elements of the available response postings. Note that the execution that spawned execution of the closure and the execution so spawned are independently atomic. The state variables of the object and any structures they reference can be changed by some other operation taken from the self-stream between the two executions. The syntactic convenience is only that: invariants that need to be preserved across independent executions need to be met at the boundaries between the execution that spawned execution of the closure and the execution so spawned.

Figure 9: CONTINUATION CLOSURES

4 Shared Variables

Shared variables are dealt with in LAMINA by treating them as references whose associated value may be mutated. A shared variable reference is constructed, accessed, and mutated by the interface operations described in this section. Support for shared data pairs and arrays is also described. For all these operations, execution is deferred and no other executions are performed by the initiating processor until the indicated operation is accomplished.[6]

Shared queues (which are streams) are also provided. These queues are maintained in a processor's local memory. When a process reads from a shared queue, it is halted and descheduled; execution is resumed when the requested data arrives.

4.1 Creating and Accessing Shared Variables

A shared variable can be allocated on a specific site (containing a processor or memory controller) and given an initial value by (`shared-variable` *value site-reference*). This creates and returns a reference to the indicated value. The *site-reference* argument is optional; if it is omitted, a randomly selected site is chosen for the default allocation. Alternatively, the construct (`in-memory` *site-reference forms*) can be used to specify a default site for all allocations done while executing the enclosed *forms*. Thus, the allocation done by the form (`in-memory` *site-reference* (`shared-variable` *value*)) is the same as that done by the form (`shared-variable` *value site-reference*).

Once a shared variable has been allocated, the following constructs may be used to access or alter its value:

- (`shared-read` *shared-variable-reference*) returns the value of the reference.
- (`shared-write` *shared-variable-reference value*) modifies the value of the reference. The new value is returned.
- (`shared-exchange` *shared-variable-reference value*) performs the same function as `shared-write`, except that the prior value of the reference is returned.

For each of these constructs, the operation is guaranteed to be completed before execution is resumed.

4.2 Shared Data Structures

LAMINA also provides support for pairs or arrays of shared variables. A structure reference is created by an executing process, which may then initialize the structure. The site for the allocation is specified by an optional *site-reference* argument, by the innermost (dynamically) enclosing `in-memory` form, or is chosen at random.

A shared pair is created by (`shared-cons` *car-value cdr-value site-reference*). The accessors for a shared pair are `shared-car` and `shared-cdr`. Pairs are altered with the forms (`shared-rplaca` *shared-pair new-car*) and (`shared-rplacd` *shared-pair new-cdr*). Also, the form (`cache-shared-pair` *shared-pair-reference*) may be used to make a local, that is, non-shared, copy of a shared pair.

The (`shared-array` *dimensions site-reference*) form returns a reference to a shared array. The *dimensions* argument is a list of positive integers, denoting the size of each dimension of the array. There are optional `:initial-element` and `:initial-contents` keyword arguments, which may be used (respectively) to initialize all the elements of the array to the single value specified or to initialize each element of the array to the value of the

[6]Note that, because the simulator is executing in a uniprocessor environment, a stack group must be maintained for each deferred execution. Thus executions must be resumable (not merely restartable) to use the shared variable LAMINA interface described below. This is discussed in section I.10.

```
(DEFUN SHARED-BUFFER (size)
  (let ((<signal> (shared-queue)) (empty? t)
        (<lock> (shared-variable t))
        (<buffer> (shared-array size :initial-element nil))
        (<head> (shared-variable 0))
        (<tail> (shared-variable 0)))
    #'(lambda (operation &optional value)
        (selectq operation
          (:insert
            (with-spin-lock <lock>
              (let* ((head (shared-read <head>))
                     (tail (shared-read <tail>))
                     (new-tail (mod (1+ tail) size)))
                (when (not (= head new-tail))
                  (shared-aset value <buffer> tail)
                  (when empty?
                    (setq empty? nil) (shared-enqueue <signal> <signal>))
                  (shared-write <tail> new-tail)))))
          (:remove
            (with-spin-lock  <lock>
              (let ((head (shared-read <head>))
                    (tail (shared-read <tail>)))
                (if (not (= head tail))
                    (let ((new-head (mod (1+ head) size)))
                      (shared-write <head> new-head)
                      (shared-aref <buffer> head))
                    (when (not empty?)
                      (setq empty? t) (shared-dequeue <signal>)))))))))
```

Figure 10: SHARED BUFFER

corresponding element in a list or a list of lists. Shared arrays are initialized to `nil` by default.

The form (`shared-aref` *shared-array-reference subscript ...*) reads elements of the shared array. The number of the subscripts supplied must agree with the dimension of the array. The form (`shared-aset` *value shared-array-reference subscript ...*) may be used to write array elements. The `cache-shared-array` function returns a local (non-shared) copy of the shared array reference it is applied to, and the `fill-shared-array` function copies data from a non-shared array into a shared array.

4.3 Shared Queues

A shared queue construct, which is implemented as a LAMINA stream, is also provided. Because queues are streams, the creator of the queue provides atomic access to the queue and when the queue is empty, maintains a FIFO queue of processes requesting data -- the requests are serviced when data is added to the queue. Further, whenever a process attempts to remove data from the queue, the process is descheduled; execution is rescheduled when the requested data arrives.

Shared queues are created by the `shared-queue` function, which takes one optional argument representing the queue's tag, which may be used for debugging. Items may be added to the queue with the `shared-enqueue` function. The `shared-dequeue` function removes and returns the top item of the queue, while the `shared-queue-top` function merely returns it.[7] A `shared-queue-p` function is also provided to test whether an item is a shared queue.

[7] In the current implementation, only FIFO queues are provided, and (in order to maintain a consistent timing model for cross address space transmissions) only shared variable or shared queue references may be placed on a shared queue.

```
(DEFUN PART4 (<array> first last)
  "Does partition on array, and returns position of pivot -- algorithm from [7.]"
  (let ((pivot (shared-aref <array> first))
        (i first) (j (1+ last)) (left-item) (right-item))
    (loop for i = (loop for ni from (1+ i) until (= ni j)
                        do (setq left-item (shared-aref <array> ni))
                        when (>= left-item pivot) return ni
                        finally (return ni))
          for j = (loop for nj downfrom (1- j) until (< nj (-1 i))
                        do (setq right-item (shared-aref <array> nj))
                        when (<= right-item pivot) return nj
                        finally (return nj)))
          if (> j i) do (shared-aset left-item <array> j)
                        (shared-aset right-item <array> i)
          else do (shared-aset right-item <array> first)
                  (shared-aset pivot <array> j) and return j))

(DEFUN MAYBE-EXCHANGE (<array> first second)
  "Exchanges first and second items, iff first is greater."
  (let ((first-item (shared-aref <array> first))
        (second-item (shared-aref <array> second)))
    (when (> first-item second-item)
      (shared-aset second-item <array> first)
      (shared-aset first-item <array> second))))
```

Figure 11: SHARED VARIABLE PARTITION & EXCHANGE

Unlike other shared variable operations, accesses to shared queues do not cause the initiating processor to stall waiting for completion. A process executing `shared-enqueue` continues immediately, without waiting for the data to arrive on the queue. A process which accesses a queue, using `shared-dequeue` or `shared-queue-top`, will be halted and descheduled. Execution is rescheduled when the data arrives, but the initiating processor may perform other executions in the meantime.

4.4 Other Synchronization

A simple spin lock is provided for busy-wait synchronization in the LAMINA shared variable interface. The form (`with-spin-lock` *shared-variable-reference form*) executes the given form after aquiring the lock specified by the indicated shared variable reference. Subsequently, the lock is released and the value produced by the execution of the form is returned. The lock must be a reference to a shared variable that was initialized to a value other than `nil`.

We might use such a synchronization operator in incrementing a shared counter as:

```
(DEFUN LOCKED-INCREMENT (<var>[8] <lock> &optional (delta 1))
  (with-spin-lock <lock>
    (let* ((value (shared-read <var>)) (new-value (+ value delta)))
      (shared-write <var> new-value))))
```

We can also create locks based on the shared queue construct. For example, we implement a mutual exclusion lock as a shared queue. To release the lock, a process places a token reference on the queue. A process acquires the lock by removing the token -- any other process which attempts to remove it will be blocked until the owner of the lock replaces the token. Alternatively, reading but not removing the token (by using `shared-queue-top`) allows

[8] By convention, we denote references to shared variables and shared queues by enclosing angle brackets, as in <lock>.

```
(DEFUN ORDER4 (<threads> <lock> requests results &optional request)
  (destructuring-bind (<array> first last) request
    (if <array>
        (let* ((pivot-position (part4 <array> first last))
               (contents (list (shared-aref <array> pivot-position)
                               pivot-position <array>)))
          (funcall              ; Order of pivot data element is established
            results :insert (shared-array 3 :initial-contents contents))
          (let ((left-diff (- pivot-position first))
                (right-diff (- last pivot-position)))
            (let ((order-left (> left-diff 2))
                  (order-right (> right-diff 2)))
              (cond
                ((and order-left order-right)     ;Order right partition
                 (let* ((request
                          (list <array> first (1- pivot-position)))
                        (request block
                          (shared-array 3 :initial-contents request)))
                   (when (null (funcall requests :insert request-block))
                     (order4 <threads> <lock> requests results request))
                   (order4 <threads> <lock> requests results
                           (list <array> (1+ pivot-position) last))))
                (order-left       ; Exchange right and then order left
                 (when (= right-diff 2)
                   (maybe-exchange <array> (1- last) last))
                 (order4 <threads> <lock> requests results
                         (list <array> first (1- pivot-position))))
                (order-right      ; Exchange left and then order right
                 (when (= left-diff 2)
                   (maybe-exchange <array> first (1+ first)))
                 (order4 <threads> <lock> requests results
                         (list <array> (1+ pivot-position) last)))
                (:else            ; Order by exhange for both left and right
                 (when (= right-diff 2)
                   (maybe-exchange <array> (1- last) last))
                 (when (= left-diff 2)
                   (maybe-exchange <array> first (1+ first,))
                 ;;Declare completion of ordering request and try again
                 (locked-increment <threads> <lock> -1)
                 (order4 <threads> <lock> requests results))))))
        (let ((<request> (funcall requests :remove)))
          (if (shared-queue-p <request>)          ;If buffer was empty...
              (if (zerop (shared-read <threads>)) ;  signal termination
                  (shared-enqueue <request> <request>)
                  (shared-queue-top <request>)    ;  or block till signalled
                  (order4 <threads> <lock> requests results))
              (locked-increment <threads> <lock>) ;Else, pick up request
              (let ((request (listarray (cache-shared-array <request>))))
                (order4 <threads> <lock> requests results request)))))))
```

Figure 12: SHARED VARIABLE ORDERING

more than one process to be resumed. This last approach more closely resembles the type of synchronization provided by signalling and waiting on condition variables in a monitor.

Figure 10 shows an example of using some of these synchronization schemes in generating a closure to perform operations on a shared buffer realized as a shared variable array. Processes first gain access to the shared array by spinning on a lock. Once access is granted, items are inserted or removed. An attempt to put information in a full buffer returns `nil` if it is unsuccessful. When an attempt is made to remove data from an empty buffer, a shared queue (rather than data) is returned -- the requesting process may then wait for something to be placed on this queue by executing `shared-queue-top`.

4.5 An Example

As an example of using the LAMINA shared variable interface, we present yet another implementation of ordering, this one using shared variables. The sets to be ordered are represented as shared arrays.

Each processor will execute an identical *thread of execution*. The execution of the thread is defined by the `order4` function, shown in figure 12. Ordering requests are distributed to the threads through a shared buffer manipulated by a closure previously formed by calling the `shared-buffer` function. A request consists of a reference to a shared array and indices representing the left and right boundaries of the array (or sub-array) to be ordered. Each thread executes in a loop as follows:

- If there is an array (or sub-array) to order, the thread partitions the sub-array, using the `part4` routine, shown in figure 11. The order of the set element used as the pivot is now established so the set element, its order, and the reference for the array (as a set identifier) is placed in the specified result queue.
- If both sub-arrays resulting from the partition are longer than two elements, the thread adds an ordering request to the queue for one sub-array and orders the other. If either sub-array has two or fewer elements, the ordering is trivial, so the thread does it (using the `maybe-exchange` function, also shown in figure 11). If neither sub-array has more than two elements, after the thread orders the sub-arrays, it signals that one less thread is currently working on any ordering requests and notes that it has no array to order.
- If the thread has no array to order, it attempts to remove a request from the queue. If successful, it signals that one more thread is trying to do ordering and orders the (sub-)array identified by the request. If the attempt is unsucessful and there are no other working threads, there will never be any more requests generated so the thread terminates. Otherwise, it tries again to remove a request from the queue. Note that the first thread to terminate places a token on the shared synchronization queue -- this wakes up the other threads, which will then terminate.

5 Utilities: Random Sites, Local Sites, Dismiss, and Boot

A few utility operations are provided by LAMINA to specify computation (and storage) sites, dismiss computations, and provide a timeout facility for applications desiring one. LAMINA also provides simulation control facilities to initiate a CARE simulation, read the current simulation time, and do a computation without increasing the simulation time.

The function `random-site` returns a reference for a site chosen randomly with uniform distribution over the processor sites in the simulated system. The function `random-memory` does the same thing over the memory controllers in the system. The function `local-site` returns a reference for the CARE site executing the function. The function `local-memory` returns a reference for a memory controller associated with the processor on which the function is executed.

In order to provide a timeout facility, the keyword `after` followed by a number of milliseconds in simulated time may be included in functions that take LAMINA keyword arguments. The simplest use might be to specify that a posting to a stream be sent at some future time.

A call to `dismiss` breaks execution. With no argument, execution is rescheduled immediately (but occurs after all previously scheduled executions are run). If an argument is specified which is a keyword, execution is terminated and will never be rescheduled. If a local stream is specified, execution is rescheduled when next that stream receives a posting -- or immediately, if that stream has a posting on it.

The current simulation time (in milliseconds) is returned by the function `simulation-time`.

Some computations in a simulated application need not (or should not) be timed. The macro (`without-clock` *form*) enclosing the forms of such computations will cause them to be accomplished "off the clock". This is generally a good idea for calls to debuggers and the like as well as for input-output operations.

Something special must be done to start up a simulation. The form

(`boot` (`at` *time site-coordinates form*) (`at`))

will spawn computations to execute forms at the indicated sites beginning at the specified times (in milliseconds). The site coordinates are given as a list, for example, '(3 2), whose length matches the represented dimensionality of the processing unit (a surface for the case shown). The `boot` construct resets the simulator and thus may only be executed as the first operation of an application being simulated.

CARE user applications should be loaded into the Zetalisp `care-user` package where all LAMINA interface constructs and primitive functions are defined.

6 Acknowledgements

The maturation of LAMINA, to the extent this has occured, has only come to pass through the sufferance of its early users. Occasionally, a user has taken a direct hand in LAMINA's definition and implementation. The work so done has invariably improved LAMINA and made it a sounder base for concurrent programming. We are indebted to our colleagues, past and present, Eric Schoen, Harold Brown, Masufumi Minami, Russell Nakano, and Max Hailperin for putting up with our offspring and helping to direct its growth.

This work takes its roots in the achievements of Daniel Friedman, David Wise, Henry Lieberman, and Carl Hewitt. The most important concepts underlying LAMINA are theirs. The distortions of those concepts, done in error or out of a preoccupation with performance (or both) are our own.

I. LAMINA Primitives

A set of functional primitives underlies the interface syntax described in the previous sections of this paper. The set of primitives described below has evolved to provide the mechanisms to support all that syntax. It is documented here so that language implementers may more easily define additional or alternative syntax.

I.1 Posting and Target Specialization

Streams acquire values as a result of postings received by them. This is directly done by the `posting` operation as in (`posting` *value* `to` *target-streams* ...). A posting may be multicast [3] by supplying a list of *target-streams*.

CARE provides a facility for specializing the values transmitted in a multicast to the individual targets of the message. Anyplace a stream is used as a target of a `posting`, it may be replaced by a `cons` of that stream and the value specialization for that stream. The value specialization will be used with the value of the posting to form a list whose elements are the list elements of the specialization (or the specification itself if it is not a list) followed by the list elements of the posting value (or the posting value itself if it is not a list). This combined list will be taken as the value of the posting when it arrives at the target stream. The simplest use of this may be to multicast some data to two remote LAMINA nodes as described in section 3, asking them to perform two different operations on the data:

```
(posting data to '((.input-stream-1 . .task-selector-1)
                   (.input-stream-2 . .task-selector-2)) ...)
```

Specialization is specified by a list of lists even if only one target is involved. This is required to distinguish it from a list of unspecialized targets.

I.2 Stream Posting Access Functions

The form (`first-posting` *stream*) returns the first posting of those present on a stream. The form (`next-posting` *stream*) does the same but removes the posting from the stream. The form (`last-posting` *stream*) returns the last posting and eliminates all others on the stream.

If the stream is empty, the three stream posting access functions, just listed, return `nil`. Otherwise, they return a posting as a list of the *value*, *clients*, *key*, *tag*, *origin*, and *properties* of the posting in that order. This list may be used with Lisp destructuring operators. Elements of this list may also be accessed by the `posting-` macros: `-value`, `-clients`, `-key`, `-tag`, `-origin`, and `-properties`. Each of these takes a posting as an argument. The number of postings available on a stream is returned by the form (`postings` *stream*).

If it is desired that execution be blocked until there is a posting for a specified stream, the stream posting access forms above may be wrapped in an (`accept` ...) construction, for example, (`accept` (`next-posting` *stream*)). When a posting is available on the indicated stream, the posting is returned to the restarted or resumed execution.

I.3 Copying Streams

A posting sent to parent streams in a tree (or graph) of streams set up by `copying` operations will result in that posting also appearing on all the descendant streams in the tree (or graph). Such a system of streams can be built by:

(`copying` *parents* `to` *child-streams* `for` *clients* ...)

The references for the *child-streams* are sent in an operation request posting to the *parents*

where they are added to the child references of the parent. The current queue of postings held in the parent stream is copied and returned in one combined posting that is multicast to the child streams. These postings become part of each child stream. When each child receives the combined postings, it sends on to the *clients* a completion posting whose value is the parent stream from which it received the posting queue. This can be used to validate that a requested copy operation has been accomplished.

I.4 Linking Streams

`Linking` is an optimization of `copying` for those cases where it is known that postings need not be retained on intermediate streams in a system of linked streams. `Linking` parent streams to child streams serves to restrict the parents to act only as intermediaries in a system of linked streams. The syntax for `linking` is:

(`linking` *parents* `to` *child-streams* `for` *clients* ...)

The references for the *child-streams* are multicast in an operation request posting to the *parents*. When a parent receives the referenc , any postings already on parent streams are sent to the children specified by the references and eliminated from the parents. Further postings are not retained on parents after they receive a `linking` directive but are immediately passed on to the child streams. For efficiency in forwarding, the implementation may bypass intermediate levels in a system of linked streams.

I.5 Value Specialization

Target specialization may also be used with the `linking` or `copying` operator to specialize the value of postings transmitted from parents to children:

(`linking` *parents* `to` '((,*child-1* . ,*value-specialization-1*)) ...)

Thereafter, all postings that traverse that link from parent to child will have the appropriate value specialization prepended to their value. The resulting value is a list whose elements are the list elements of the value specialization (or the value specialization itself if it is not a list) and the list elements of the posting value (or the posting value itself if it is not a list). This is the mechanism used to support the syntax of `with-postings` when a continuation closure with associated response posting are to be put on a the self-stream of an object.

I.6 Relocating Streams

A `linking` operation does not change the way that a child stream orders postings or presents them. Relocating a stream from one site to another with that stream's means of ordering and presenting postings (together with any accumulated postings) is specified by:

(`relocating` *parents* `to` *child-streams* `for` *clients* ...)

This is used when there is an attempt to read from a stream that is not local to a site. The attempt causes the reference used to specify that the target stream target a new child stream, the relocation of the previously specified target. No change can be detected in the operation of `reference-eq` on the reference after relocation.

I.7 Group Streams

An application in LAMINA may wish to view a group of streams as a composite, a ***group-stream***, carrying out some operation when all of the streams in the group have received a posting. To minimize unproductive scheduling, computations may wait on such stream composites rather than the individual streams. Group-streams are created by `new-stream` called with a `:group` keyword argument as in: (`new-stream` *tag* `:group` *member-streams*). A future, that is a stream

which may have at most one value, may be a member of many groups but otherwise a stream may be the member of only one group. If such streams of values are to be made available to several groups, a system of linked or copied streams can be created as discussed previously.

If a member stream is not local to the site of its group stream, a local member stream is created and the remote member stream is relocated there. The postings sent to the local member streams are taken from the member streams whenever a request that has been made to accept a posting from a group stream can be satisfied. Each posting available from a group stream will contain a list of postings received by its component streams as its value.

The order of posting elements in the list representing a group stream posting will correspond to the order indicated in specifying the component streams of the group stream when it was formed by calling the function `new-stream` as shown above.

Group streams are used to implement `with-postings` constructs. Continuations are only scheduled when values are available on all the streams included in the specified stream bindings.

I.8 Accessing and Exchanging Stream Values

Posting-by-posting access of the information on streams may be accomplished by requesting that a stream access function be applied to the streams at the site they exist on:

(`accessing` *access-function* `on` *target-streams* `for` *client-streams* ...)

The *access-function* may be any of the stream posting access functions, for example, the function `next-posting` described previously. A posting will be sent to the client streams when one is available on a target stream. This is the only way provided for expressing competitive access to a common stream.

An interlocked operation on streams is provided:

(`exchanging` *value* `on` *target-streams* `for` *client-streams* ...)

This causes `last-posting` to be applied to each target stream and the result sent to each client stream. The *value* replaces the last posting on the target stream. This is done atomically with applying `last-posting` to the stream.

I.9 Spawning a Restartable Computation

A separate, concurrent computation is created by spawning the execution of a closure as shown in the following example:

(`spawning #'(lambda () form) on` *site-reference* `for` *clients* ...)

The closure is formed and the *clients* returned immediately as the value of the `spawning` operation. The closure will sent to the indicated site and eventually executed there. The result of that execution will be returned to the specified client streams.

Spawned computations can block waiting for a value to be available on a stream. When the value is available they will be restarted and any intermediate computations done previously will be redone. This approach is taken to avoid creation of stack groups for every spawned computation. Resumable (as opposed to restartable) computations with their own stack groups can be created by LAMINA operations discussed in section I.10.

As an alternative to mounting computations with their own stack groups, the continuations of partially completed computations can be spawned on the same site as their parent. This is done by the `with-values` functional programming interface constructs described in section 2 and by the `with-postings` object-oriented programming interface constructs described in section 3.6.

I.10 Mounting Executions with Stack Groups

If an execution is blocked on trying to `accept` something from an empty stream, it is either restarted (as discussed above) or resumed when that stream receives a posting. In general, resuming a computation from where it left off (without spawning continuations) requires preserving indeterminate amounts of intermediate state with a stack group. Maintaining many independent stack groups is certainly an expensive operation in simulation and may also be so in a target system implementation.

However, for occasions when the full power and expense of stack group switching is warranted, LAMINA provides a construct in the same format as `spawning`:

(`mounting` *closure* `on` *site-references* `for` *clients*...)

The clients are returned immediately. The closure is sent to the specified site(s) where it will be applied and the computed result sent to the clients. Note that the `boot` operation discussed in section 5 spawns rather than mounts a computation. If a mounted computation is needed, it must be explicitly mounted by the computation that `boot` spawns.

One could implement a multiple fork and join construct (like `cobegin ... coend`) by mounting a number of processes with a common client stream. The creator could then wait for the appropriate number of responses on the client stream (to insure that the other processes had completed) and then continue its execution.

In applications that wish to view executions created with `mounting` as non-terminating, the execution will typically have an initial section that sends a reference for a newly created (task) stream to mutually agreed upon streams (by an explicit `posting`). The referenced task stream will then be used to supply the newly mounted execution with additional operations to perform after it completes its starting procedures.

I.11 Loading Sites and Passing Arguments to Remote Closures

An item may be sent to a remote site, a reference for it created there, and the reference sent to specified clients:

(`loading` *item* `on` *site-reference* `for` *client-streams* ...)

The *client-streams* are returned immediately by the form. Remote closures may be created by `loading` closures:

(`loading #'(lambda` *arglist form*) `on` *site-reference* `for (new-stream)` ...)

The new stream immediately returned will eventually get a value representing a reference for the closure on the specified site. A remote closure may be applied to locally evaluated arguments by passing it those arguments:

(`passing` *arglist* `to` *closure-reference* `for` *clients* ...)

The result of the remote application is sent to the specified clients. The `loading` and `passing` operations are combined in `spawning`.

II. LAMINA Primitives and Interfaces

LAMINA primitive and interface functions are listed in this appendix with a reference to the section or sections in which they are described and discussed.

II.1 References

1.3	REFERENCE *item*	*Function*
1.3	REFERENCE-SITE *reference*	*Function*
1.3	REFERENCE-EQ *reference1 reference2*	*Function*

II.2 Functional Programming Interface

2	FUTURE *form*	*Macro*
2	WITH-VALUES *future-bindings* &body *forms* The *future-bindings* is a list each element of which is itself a list: (*binding-pattern future-specifier*).	*Macro*

II.3 Object Oriented Programming Interface

3.1	SENDING *self-streams task-selector value* &rest *lamina-keywords*	*Function*
3.2, I.7	NEW-STREAM &optional *tag* &key group *member-streams*	*Function*
3.2	NEW-FUTURE &optional *tag*	*Function*
3.2	ORDERED-STREAM &optional *tag*	*Function*
3.2	SEQUENCED-STREAM &optional *tag*	*Function*
3.3	LAMINA, ORDERED-SELF-STREAM, and SEQUENCED-SELF-STREAM	*Flavors*
3.3	SELF-STREAM of LAMINA	*Instance Variable*
3.4	DEFTRIGGER (*object-type task-selector*) *trigger-pattern* &optional *documentation-string* &body *forms* The trigger pattern destructures the list (*value clients key tag origin properties*).	*Macro*
3.5	CREATE-SELF-STREAM *object-type* &optional *tag*	*Function*
3.5	CREATING *object-type state-variable-settings* &rest *lamina-keywords* *State-variable-settings* is a list alternating (state-variable) keywords and values.	*Function*
3.6	WITH-POSTINGS *stream-bindings* &body *forms* The *stream-bindings* is a list each element of which is itself a list: (*binding-pattern stream-specifier*).	*Macro*

II.4 Shared Variable Interface

4.1	SHARED-VARIABLE *site-reference value*	*Function*
4.1	IN-MEMORY *site* &body *forms*	*Macro*
4.1	SHARED-READ *shared-variable-reference*	*Function*
4.1	SHARED-WRITE *value shared-variable-reference*	*Function*
4.1	SHARED-EXCHANGE *value shared-variable-reference*	*Function*
4.2	SHARED-CONS *car-value cdr-value* &optional *site-reference*	*Function*
4.2	SHARED-CAR *shared-pair-reference*	*Function*
4.2	SHARED-CDR *shared-pair-reference*	*Function*
4.2	SHARED-RPLACA *shared-pair-reference new-car*	*Function*
4.2	SHARED-RPLACD *shared-pair-reference new-cdr*	*Function*
4.2	CACHE-SHARED-PAIR *shared-pair-reference*	*Function*
4.2	SHARED-ARRAY *dimensions* &optional *site-reference* &key :initial-element *value* :initial-contents *value-sequences*	*Function*
4.2	SHARED-AREF *shared-array-reference* &rest *subscripts*	*Function*
4.2	SHARED-ASET *value shared-array-reference* &rest *subscripts*	*Function*
4.2	CACHE-SHARED-ARRAY *shared-array-reference*	*Function*
4.2	FILL-SHARED-ARRAY *array shared-array-reference*	*Function*
4.3	SHARED-QUEUE *tag*	*Function*
4.3	SHARED-ENQUEUE *reference shared-queue-reference*	*Function*
4.3	SHARED-DEQUEUE *shared-queue-reference*	*Function*
4.3	SHARED-QUEUE-TOP *shared-queue-reference*	*Function*
4.3	SHARED-QUEUE-P *item*	*Function*
4.4	WITH-SPIN-LOCK *shared-variable-reference* &body *form*	*Macro*

II.5 Utility Operations

5	RANDOM-SITE and RANDOM-MEMORY	*Functions*
5	LOCAL-MEMORY and LOCAL-SITE	*Functions*
5	DISMISS &optional *stream-or-keyword*	*Function*
5	SIMULATION-TIME	*Function*
5	WITHOUT-CLOCK &body *forms*	*Macro*
5	BOOT &rest *at-forms* An *at-form* is a list of the form: (at *time site-coordinates* &body *forms*)	*Macro*

II.6 Primitives

I.1	POSTING *value* &rest *lamina-keywords*	*Function*
I.2	POSTINGS *stream*	*Function*
I.2	FIRST-POSTING *local-stream*	*Function*
I.2	NEXT-POSTING *local-stream*	*Function*
I.2	LAST-POSTING *local-stream*	*Function*
I.2	POSTING-VALUE *posting*	*Function*
I.2	POSTING-CLIENTS *posting*	*Function*
I.2	POSTING-KEY *posting*	*Function*
I.2	POSTING-TAG *posting*	*Function*
I.2	POSTING-ORIGIN *posting*	*Function*
I.2	POSTING-PROPERTIES *posting*	*Function*
I.2	ACCEPT *stream-access-form*	*Macro*
I.3	COPYING *parent-streams* &rest *lamina-keywords*	*Function*
I.4, I.5	LINKING *parent-streams* &rest *lamina-keywords*	*Function*
I.6	RELOCATING *parent-streams* &rest *lamina-keywords*	*Function*
I.8	ACCESSING *access-function* &rest *lamina-keywords*	*Function*
I.8	EXCHANGING *value* &rest *lamina-keywords*	*Function*
I.9	SPAWNING *function* &rest *lamina-keywords*	*Function*
I.10	MOUNTING *function* &rest *lamina-keywords*	*Function*
I.11	LOADING *item* &rest *lamina-keywords*	*Function*
I.11	PASSING *arglist* &rest *lamina-keywords*	*Function*

References

1. Gregory R. Andrews and Fred B. Schneider. "Concepts and Notations for Concurrent Programming." *Computing Surveys 15*, 1 (March 1983), 3-43.

2. Harold D. Brown, Eric Schoen, and Bruce A. Delagi. An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures. Tech. Rept. STAN-CS-86-1136 or KSL-86-69, Stanford University, October, 1986.

3. Gregory Byrd, Russell Nakano, and Bruce Delagi. A Dynamic Cut-Through Communication Protocol with Multicast. Tech. Rept. KSL-87-44, Knowledge Systems Laboratory, Stanford University, August, 1987.

4. Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. An Instrumented Architectural Simulation System. Tech. Rept. STAN-CS-87-1148 or KSL-86-36, Knowledge Systems Laboratory, Computer Science Department, Stanford University, January, 1987.

5. Daniel P. Friedman and David S. Wise. An Indeterminate Constructor For Applicative Programming. 7th Annual Symposium on Principles of Programming Languages, 1980, pp. 245-250.

6. Robert H. Halstead, Jr. "Multilisp: A Language for Concurrent Symbolic Computation." *ACM Transactions on Programming Languages and Systems 7*, 4 (October 1985), .

7. Donald E. Knuth. *The Art of Computer Programming (Volume 3)*. Addison-Wesley, Reading, Massachusetts, 1973.

8. Henry Lieberman. Thinking About Lots of Things Without Getting Confused. AI Memo 626, MIT, May, 1981.

9. David A. Moon. Object Oriented Programming with Flavors. Object-Oriented Programming Systems, Languages, and Applications [OOPSLA] '86 Proceedings, September, 1986, pp. 1-8.

10. Russell Nakano and Masafumi Minami. Experiments with a Knowledge-Based System on a Multiprocessor. Tech. Rept. KSL-87-61, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1987.

11. Ehud Shapiro. "Concurrent Prolog: A Progress Report." *Computer 18* (August 1986), 44-58.

12. Steele, G.L. Jr. Lambda, the Ultimate Declarative. AI Memo 379, MIT, November, 1976.

13. Guy L. Steele. *Common Lisp: The Language*. Digital Press, Billerica, MA 01862, 1984.

14. Daniel Weinreb and David Moon. *Lisp Machine Manual*. Symbolics, Cambridge, MA, 1981.

Knowledge Systems Laboratory
Report No. KSL 86-36

January 1987

An Instrumented Architectural Simulation System

by

Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

WORKSYSTEMS ENGINEERING GROUP
Low End Systems and Technology
Digital Equipment Corporation
Maynard, Massachusetts 01754

This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Greg Byrd was supported by an NSF Graduate Fellowship and by the Stanford University Department of Electrical Engineering.

ABSTRACT

AN INSTRUMENTED ARCHITECTURAL SIMULATION SYSTEM

Simulation of systems at an architectural level can offer an effective way to study critical design choices if (1) the performance of the simulator is adequate to examine designs executing significant code bodies -- not just toy problems or small application fragments, (2) the details of the simulation include the critical details of the design, (3) the view of the design presented by the simulator instrumentation leads to useful insights on the problems with the design, and (4) there is enough flexibility in the simulation system so that the asking of unplanned questions is not suppressed by the weight of the mechanics involved in making changes either in the design or its measurement. A simulation system with these goals is described together with the approach to its implementation. Its application to the study of a particular class of multiprocessor hardware system architectures is illustrated.

1 INTRODUCTION

Simulation systems are quite often developed in the context of a particular problem. To a degree, this is true for SIMPLE, an event based simulation system, and CARE, the computer array emulator that runs on SIMPLE.[1] The problem motivating the development of both SIMPLE and CARE was the performance study of 100 to 1000-element multiprocessor systems executing a set of signal interpretation applications implemented as "1000 rule equivalent expert systems" [2].

A set of constraints pertinent to this problem governed the design of SIMPLE/CARE. The applications represented significant bodies of code and so simulation run times were expected to be an important consideration. Moreover, the issues involved with the interactions of multiprocessor system elements were sufficiently unexplored prior to simulation that simplifications in the CARE system model, specifically with respect to element interactions, were suspect. This need for detail was, of course, in tension with the need for simulation performance. The ways that simulated system components would be composed into complete systems was initially difficult to bound. Further, it was clear that the models of these components would be elaborated over time and would undergo substantial change as design concepts evolved. It was also clear that the ways of examining the operation of these components would change independently (and at a great rate) as early experience indicated what alternative aspect of system operation *should* have been monitored in any given completed run.

The design goals that emerged then were (1) that the simulation system should support the management of substantial flexibility with regard to simulated system structure, function, and instrumentation and (2) that, in order to accomplish runs in acceptable elapsed times, the detail of simulation should be particularly focused on the communications, process scheduling, and context switching support facilities of the simulated system -- that is, on just those aspects of system execution critical to multiprocessor (as opposed to uniprocessor) operation.

1.1 Design Time Interaction And Run Time Operation

Encapsulation of the state of design components with the procedures that manipulate that state is one clear way to manage design evolution. Such encapsulation partitions the design along well defined boundaries. Components (by and large) interact with other components only through defined *ports*. Connections between components terminate at such ports. When a system simulation is initialized, connections are traced so that for every port, the simulator knows the connected (terminating) ports together with their containing components. Once such initialization is complete, that is, throughout the simulation run, assertions about the state of a port of one component can be directly translated to assertions about the state of connected ports of other components.

Partitioning issues of system **structure**, **component behavior**, and **instrumentation** into separate domains of consideration helps in managing a design that is both fluid and complex. System **structure**, that is, the relationship between components, can be specified through use of an interactive, graphics structure editor and is largely independent of component function per se. **Component behavior** is encapsulated in a set of definitions pertinent to the given class of component. Each component in a SIMPLE simulated system is a member of a class defined for that component type. **Instrumentation** is automatically and invisibly made part of the definition of each simulated component that is to be monitored during a run. This is done by arranging that the class of every component to be monitored is a specialization of the general *instrumented-box* class. The basic data structures and procedures for monitoring simulated components and maintaining the organizational relationships between each component and its related instrumentation are inherited through this general, ancestral class and are thus made a separate, substantially independent consideration in the design.

[1] SIMPLE and CARE were developed by the authors at the Knowledge Systems Lab of Stanford University. SIMPLE is a descendent of PALLADIO [1] optimized for the subset of PALLADIO's capabilities relevant to hierarchical design capture and simulation. It is written in Zetalisp [4] and currently runs on Symbolics 3600 machines and TI Explorers

A further partitioning of concerns is employed to separate out the definition of the application programming language interface and its support (as provided by CARE) from the underlying information flow control governing component behavior. The behavioral descriptions of components (which are expressed as sets of condition/action rules) deal generically with gating information, independently of the structure of the information, between ports of the component and its internal state variables. This is separated in the component model definitions from the functions performed to create and manipulate the information so gated. The simulated implementation of the application programming language support facilities, on the other hand, relies only on the specifics of the information and its structure and plays no part in gating it between the components of the system. Changing the definition of the application language is thus done independently of changing component flow control behavior. The application programmer and the implementer of the application language interface may use whatever data structures seem suitable to them, be they numbers and keywords or procedure bodies and execution environments. The simulation system doesn't care.

The *component probe* definitions, that is, the specifications of what information should be captured for each component type, are separated from the descriptions of the behavior of such components. In designing for flexibility in the instrumentation system, it turned out to be important to further divide the information presentation from the information collection issues. The mapping from particular component probes to particular *instrument panels* and the transformations to be applied to the information as it passed from a given kind of probe to a given panel (and between panels) is captured in the *instrument specification.* This is a definition of what kinds of panels are included in an *instrument*, how they fit on an *instrument screen*, how they are labeled and scaled, and what information from which kinds of probes are displayed on each panel. The instrument specification also indicates what kinds of probes are to be connected to which kinds (that is, which classes) of components in the system.

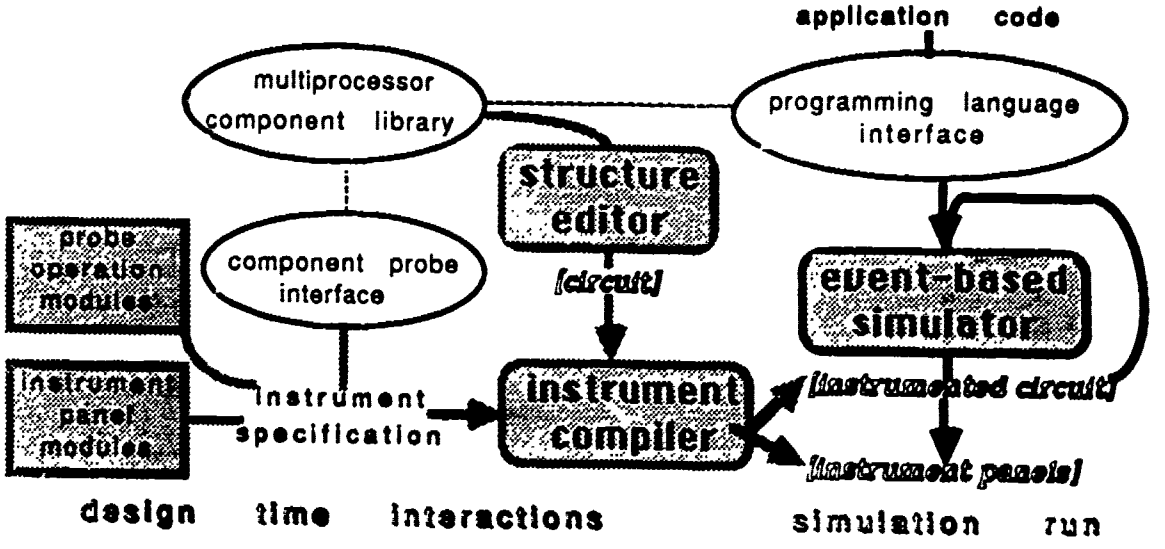


Figure 1: Design Time Interactions and Run Time Representations

Putting together all the definitions of components, component probes, panels, instruments, applications interfaces, and inter-component relationships is done in a set of design time interactions by a system architect. These interactions are used by the simulation system to generate efficient run time representations so that simulation performance goals can be met. Figure 1 illustrates the partition between design time interactions and simulation run time operation. Structure editing pulls together components from the component library to produce a *circuit*. Associated with some components in the library, there are definitions for the syntax and underlying mechanisms of a multiprocessor applications language. These specify the

interface used to provide the program input to the multiprocessor system being simulated.[2] The definitions used to generate component probes are associated with each library component to be monitored. There may be several such definitions, each appropriate to measuring a different aspect of the associated component's operation. An instrument specification selects from these definitions, elaborates them with selections from a set of *probe operation modules* to include any pre-processing (for example, a moving average) to be calculated by the probe, and indicates under what conditions what information from the probe is to be sent to which panels of the instrument and how it is to be transformed and displayed there. Instrument specifications also partition the screen among the panels of the instrument. The end product of these design time interactions is an *instrumented circuit* and an *instrument*. The instrument comprises a set of instrument panels and a set of constraints relating them to the instrument screen. The instrumented circuit ties together instances of components, probes, and panels for a simulation run.

For each defined class of component and its associated probes, the design time interactions produce code bodies that accomplish simulation operations during a run. It is an attribute of the underlying Lisp base of the simulation system that changes in these definitions have immediate effect even during a simulation run -- an important capability during debugging.

2 STRUCTURE AND COMPOSITION

Design time interactions to specify a system include the establishment of component relationships. Such specifications can be said to accomplish the composition of the system from its components and so define its structure. SIMPLE supports hierarchical composition: components may be described in terms of a fixed set of relationships among their sub-components. Additionally, such composite components may have function beyond what can be inferred strictly from their composition. All this can then be included a higher level composite (as shown in figure 2) and so on indefinitely until the top level "circuit", the system structure, is reached.

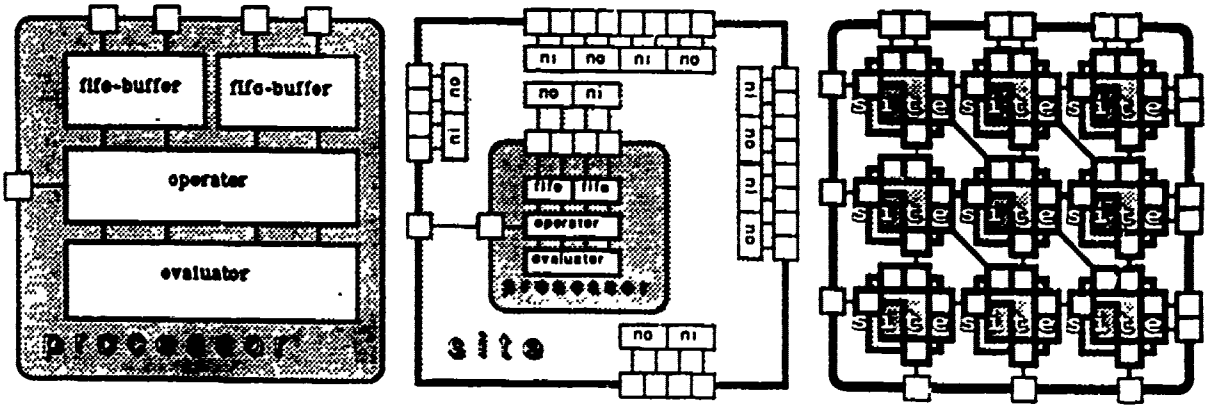


Figure 2: Hierarchical Composition

The behavior induced on a composite component from its parts changes according to the behavior of its parts. Thus, for example in figure 2, if at any time during a simulation the function of CARE *operator* components is changed by redefining their operation, the behavior

[2]The language primitives supplied can be used to define multiprocessor language interfaces for either shared-variable or value-passing paradigms. As supplied, the language interface built on these primitives supports value-passing on streams between objects but alternative interfaces can be (and have been) easily defined in terms of the given primitives.

of the nine-site grid is in immediate correspondence.[3]

Composition is described graphically and interactively in SIMPLE by picking a previously specified component type from a menu, placing it in relationship to other components with "mouse" movements, and, through the same means, specifying the connections between its selected ports and those of other components (as indicated in figure 3).

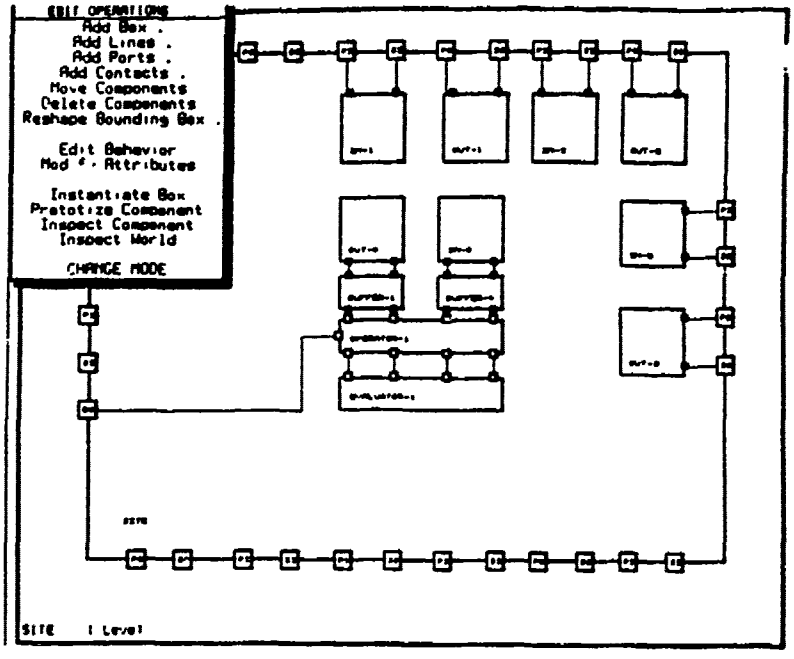


Figure 3: Graphic Structure Specification

Through another menu selection, ports can be defined for the new composite component so that it, in turn, can be fitted into yet higher level structures. Such external ports can be connected directly to ports of sub-components "within" the composite. If this is done, information appearing on that external port will be the responsibility of the connected sub-component. By this same means, a component previously described as a base level component, can be redefined as a composite of yet lower level elements as its design is elaborated with further details.

Components and (internal) connections can also be deleted from a library component and replaced with substitute components. After all sub-components and connections have been added, deleted, elaborated, and replaced as required, the completed structure can then be entered into a library of components and used in turn to compose higher or equivalent level components.

2.1 CARE Base Components

CARE supplies a small library of system level base component types. Currently these are the *net-input*, the *net-output*, the *fifo-buffer*, the *operator*, and the *evaluator*. The net-input, net-

[3]However, for reasons concerning simulation performance and because of their relatively low frequency, changes in the number and names of the internal state variables of components and the structural relationships between sub-components of a composite are not reflected in an already instantiated circuit. Changes in the internal structure of a CARE *site* library component, for example, will be reflected only in circuits instantiated after the change took effect. For this reason and to reduce long term storage requirements and load time for the fundamentally iterative circuits that we primarily study, we do not keep files of instantiated circuits. They are instantiated as needed from a high level library component with the same prototypical structure.

output and fifo-buffer accept (or block), route, and buffer transmissions. They do so in accordance with a dynamic, flow-controlled, multicast, cut-through communications protocol as described in [3]. The evaluator does the real work of the application: evaluating the application of functions to their parameters. The operator does the overhead work associated with such evaluations: for example, scheduling processes and sending and receiving (but not routing) messages.

In keeping with the objective of focusing simulation cycles on the aspects of the simulation particularly relevant to multiprocessor operation, the behaviors of the net-input, net-output, and fifo-buffer component classes are defined in fair detail, that is, at the register transfer level. Routing operations are described procedurally and assumed to occur within a time set by a parameter to the simulation. As indicated previously, the simulation of the operator and evaluator is broken into two aspects: the control of the flow of information and the functions performed on that information. The former is described in terms of SIMPLE behavior rules (as documented in section 3), register transfer by register transfer. The latter is described directly in terms of procedures and the simulated time taken by such procedures is modeled. In the case of the operator, this is done as a function of the number of storage cells manipulated during an operator procedure. In the case of the evaluator, this is done as a function of the execution time used by the machine executing the simulation, that is, the simulation vehicle.

2.2 CARE Composite Components

The prototypical composite component supplied with CARE is the *site*. As supplied, it includes net-inputs and net-outputs for up to eight "neighboring" components (generally other sites), a net-input and a net-output with associated fifo-buffers for local receptions and transmissions, and, finally, an operator and evaluator as described above. Specializations of the site, for example, the *torus-site*, exist in the library to fit the site into alternative topologies by supplementing the site routing and wiring procedures as appropriate to the topology.

2.3 Automatic Composition in CARE

Although any connection of components can be created by the means noted previously, for some repetitive, well patterned systems of connections, composition can be automated. The CARE library includes a component, the *iterated-cell*, which represents a template for the creation of composite components by iteration of a unit cell. The unit cells (for example, the torus-site) are specializations of other components (for example, the site) as just discussed. The specializations include a method for responding to a request to provide a wiring list. Such a list associates each source port of a cell with the corresponding destination port (in terms of port names) and the position of the destination cell relative to the source cell in the iterated structure. The iterated cell component uses this information to make the required connections between each of its constituent cells.

3 SPECIFYING BEHAVIOR

SIMPLE is an event based simulator. The behavior of a simulated component is described in terms of responses to the events pertinent to that component. A component's response may include consequent events to be handled by the simulator as well as direct operations on component state. Assertion of consequent events and the responses to them (involving further consequences) drives the simulation. When there are no more events to handle, the simulation is complete.

To maintain modularity in a simulation system, responses to simulation events should be local to the affected component and its defined ports, that is, its connection to the remainder of the simulated system. The composition system of the simulator maintains the relationship between ports of one component and those of other components connected to them. Assertions

relative to a port of a component are thus systematically translated to events pertinent to components connected to it. This is the general mechanism for event propagation between components. In a limited number of cases, a direct operation on a related component may be appropriate. With fair warning about its possibility of abuse, a facility is provided to accomplish this.

3.1 Behavioral Rules

The behavior of a component is described in terms of its responses to pertinent events. Each event stipulates the component affected, its port or state variable signalled with an assertion, the asserted value, and the simulated "time" of the event. The time of an event may be thought of as the "current" simulation time. Differences in event times represent the temporal relationship between events. Event times in SIMPLE simulations are monotonically increasing.

For each type of component, there is a procedure to handle pertinent events. The arguments to the procedure are those stipulated by the event (as just described). The procedure tests for conditions and, as satisfied, asserts or directly effects consequent actions. The conditions may include arbitrary predicates on the event parameters and the state variables of the component.

Event based simulators are based on the assumption that state and port variables remain unchanged until explicitly modified. Synchronous designs, that is, those in which the opportunities for state change are temporally quantized to a clock, can be modeled in such implicitly asynchronous, event based simulators by asserting the clock signal on a port of each and every clocked component of the simulated system. If only some of the components in a system need take action on each clock signal, there is an obvious inefficiency in this approach that is crippling for systems with even a modest number of components.

If, however, event times in an event based simulator are restricted to integers, the clock can be assumed. All that is needed is a way to detect the event for which a boolean combination of conditions as strobed by an assumed clock is first met. Primitive condition predicates are supplied for detecting an "edge" (a value changed by the current event) with a coincident "level" (a value set before the current event) of two ports or state variables of a component in either of the two possible event sequences. The predicate `both-states` in the example evaluator behavior rule shown in figure 4 has these semantics.

```
;;If the evaluator is ready and there is at least one runnable process...
((or (both-states Evaluator-Status[4] 'ready Evaluator-Queue-Status 'some)
     (both-states Evaluator-Status 'ready Evaluator-Queue-Status 'full))
;;... make it current, start evaluation, and adjust status as per removal.
 (setq Evaluator-Status 'busy)                          ;block rule
 (assert-state Evaluator-Status 'busy now)              ;next event
 (setq Current-Evaluation (queue-take Evaluator-Queue)) ;note process
 (user-evaluate Current-Evaluation now)                 ;execute it
 (send self :evaluator-queue-decreased now))            ;note change
```

Figure 4: Example Condition/Action Behavior Rule

Figure 4 illustrates the generality of SIMPLE behavioral descriptions. The underlying object-oriented programming system, Flavors [4], in which SIMPLE is implemented provides for direct reference of component state variables. The conditions and actions of behavior rules for a component then need only name the component's port or state variable (as stipulated in the definition of that component type) to get or change the appropriate value in the component instance for which the event is pertinent. Actions may include arbitrary procedures: for example, the procedures `user-evaluate` and `queue-take` in the given example.

[4] By convention, component state variables are written in capitalized form.

3.2 Using Methods

The environment for the execution of the procedures defining responses to events includes the state variables and ports of the component instance for which the event is pertinent. These procedures are Flavor *methods* [4] (in this case corresponding to the `:ApplyRules` message) of the component type and, as just noted, refer implicitly to the state variables of the component instance handling the event. Other methods may be defined for simulated components: for example, the `:evaluator-queue-decreased` method invoked in figure 4. Such methods have proved to be a natural way to realize the functional operations of components not described by behavior rules.

The composition system leaves information about the enclosing and contained component instances for each simulated component in system defined state variables of that component. With this information, methods directly referencing the ports and state variables of such related components may be invoked as needed. This is a useful but sharp-edged facility. The warning about loss of modularity given previously applies here.

4 INSTRUMENTATION

The results of a simulation are primarily the insights it provides into the operation of the simulated system. The "insight" we frequently experienced using an early version of the simulation system was that more interesting results could have been produced by the run just completed if only the instrumentation had been different. With this in mind, the design for the current version of the simulation instrumentation system was aimed at flexibility. This was attained without significant performance impact by building efficient run-time system structures before each run, as outlined in section 1.1, from the declarations defining the instrumentation.

The organization of the instrumentation system is pictured in figure 5. The simulator interacts with component instances through assertions, that is, calls on an `assert` function, in behavior rules (the methods associated with `:ApplyRules` messages). All instrumented components are specializations of an *instrumented-box* (as well as other classes). After each invocation of `:ApplyRules` for such components, the `:ApplyRules` method for a generic instrumented-box is applied. This causes invocation of the `:trigger` method for each *component-probe* associated with that component. Since this flow of measurements is accomplished by means invisible to the the writer of behavior methods for a component, the concerr surrounding component design are effectively partitioned from component instrumentation. The remainder of this section details these "invisible" means used to accomplish measurement flow during a simulation run as the measurements are staged from components through component probes to instrument panels.

4.1 Component Probes

The first filtering of events is done by component probes. Some events cause no further measurement activity since, as it turns out, not all events merit action on the part of the instrumentation system. The parameters of the event and the ports and state variables of the instrumented component dealing with the event are available to the component probe as are the state variables of the probe itself. Each piece of the selected information is tagged with an identifying keyword and passed along as the parameters of the `:trigger` method along with a keyword identifying the type of component probe, a number representing the current event time, and a pointer to the component with which the information is to be associated in the display. This pointer might be to some component related to the one actually handling the event, for example, the component enclosing it.

Component probes may be composed of predefined probe operation modules to do standard calculations (for example, moving averages) and then to forward the results to selected panels. In order to automate the composition of probes to accomplish such operations, each of these operations is chained together by invoking the method for that probe that is associated with

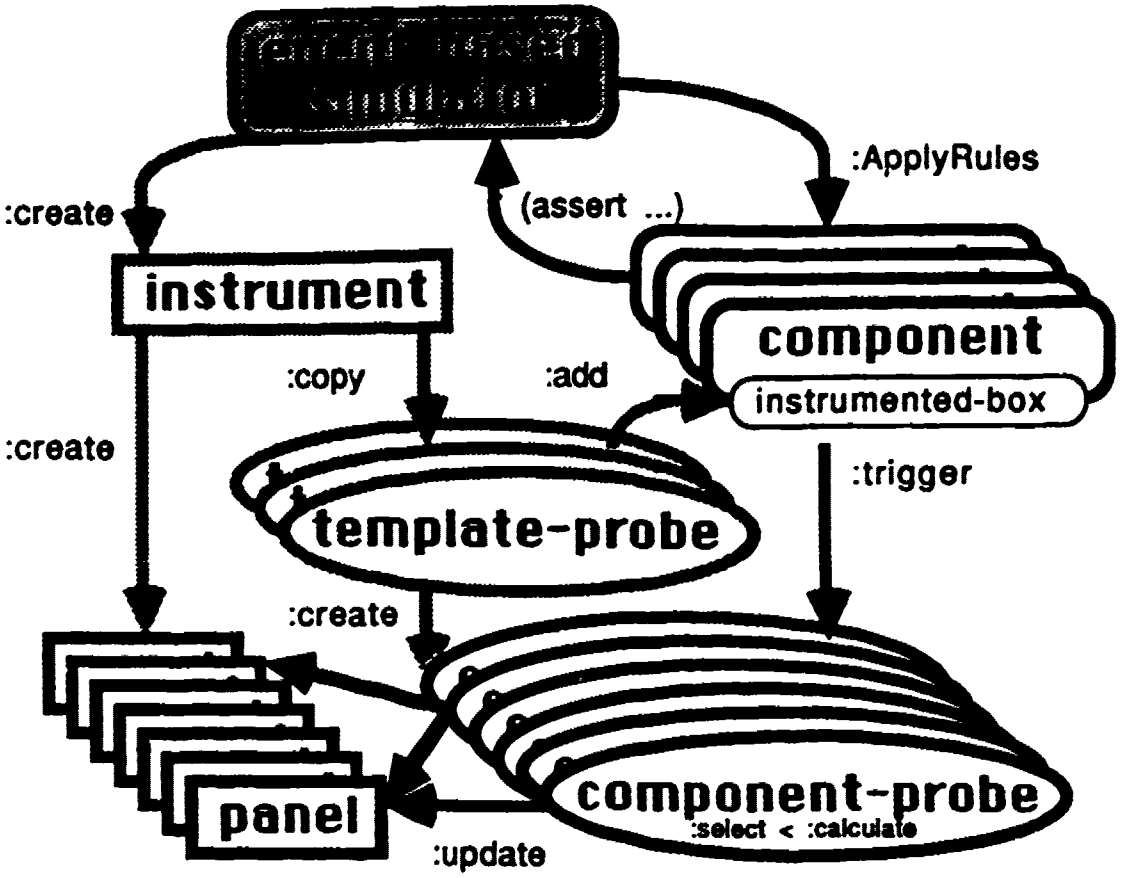


Figure 5: Instrument System Organization

the system-defined message name of the generic next operation. Thus, the `:trigger` method calls the `:calculate` method of the probe which, in turn, calls its `:select` method which, finally, calls the `:update` method of the selected panels associated with the probe. Probes are composed by naming them as specializations of appropriate probe operation modules (for example a `:calculate` module for moving averages) as desired. The default, if no specializations are stipulated, is to pass through information without change to all the panels associated with a probe.

Information flow between components and panels is accomplished by the component probes associated with each instrumented component. The creation of such component probes and their association with appropriate components (by execution of `:add` methods) accomplishes the instrumentation of a circuit. This is done when an instrument is created. During simulation initialization, the components of the circuit (and their sub-components) to be instrumented are (recursively) examined by each *template probe* defined for the instrument to see if they are to be monitored. If so, the `:copy` method for the given template probe is invoked to create a new instance of the appropriate component probe and add it to the probes connected to the component. Each template probe previously received the identifiers for the panels to which its clones should send information. These will be the panels identified when a component probe invokes the `:update` method.

4.2 Instrument Specifications

The operations performed by an instrument panel are to:

- *Find* information previously stored according to the component pointer supplied by the `:update` method;

- *Link* new data structures as needed (to save such information) to other such structures of the panel;
- *Save* in these data structures the results of expressions that reference indicated keyed information from the :update parameters and the prior contents of the structures;
- *Send* the results of periodic analyses on the information associated with a panel for display by the same panel or by some other; and
- *Show* processed information in the manner specified for the panel.

The defaults for the panel operations supply the most commonly required specifications implicitly, so simple operations are simply specified. These defaults can be overridden as needed and either predefined or user specified alternatives for the panel operations can be selected in their place. Arbitrarily complex (Lisp) expressions can be used to specify the transformations between the information provided by a probe and that saved and displayed by the panel.

These transformations and all the default overrides for the panel operations that are stipulated in the instrument declaration are scanned when a new instrument is created for a simulation session. They are compiled at that time into code bodies referenced by run time control blocks associated with each panel. A simulated system is instrumented by examining all of its components and attaching to each component the copies of template probes specified by the instrument definition that are appropriate for the component (by means of calls on the :copy and :add methods for the probe). This can be a many to many relationship as shown in figure 6.

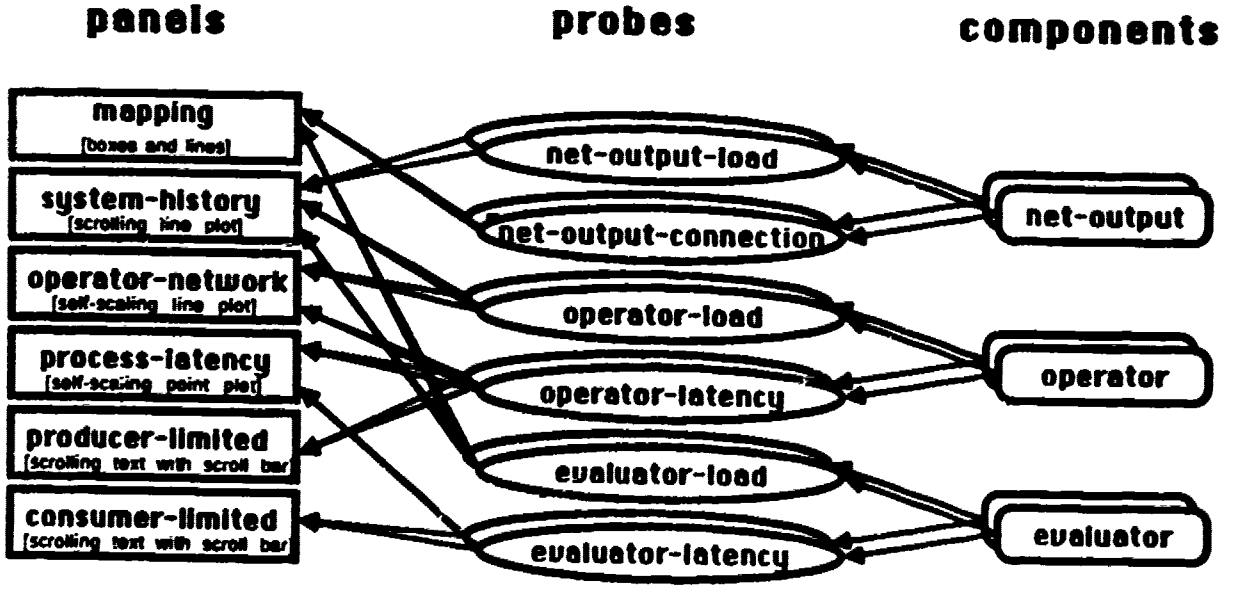


Figure 6: Instrument Probe and Panel Relationships

Component probes to measure "load" and "latency" are specified in the given example for each operator and evaluator in the circuit. The "load" and current "connection" for each net-output is also to be monitored. Some panels, for example the one showing "consumer-limited" processes, receive inputs from only one type of component probe, those measuring evaluator latency. Others, such as the one measuring "process-latency" receive inputs from more than one kind of probe (in this case, from probes measuring operator latency as well as those measuring evaluator latency). A way must thus be provided to distinguish the type of probe sending information to a panel; this is described in the next section.

Some probes send information to only one panel, for example, the net-output connection probes. Others monitor information which is needed by several panels, for example, the operator latency probe. Transformation of the raw information provided by a probe will need to be specialized to the information expected by each panel receiving it. A general way to stipulate these transformations is stipulated in the next section.

5 EXAMPLE PANELS

Some example panels are described in this section to give a feel for the instrumentation possibilities available in CARE and elaborate on how the requirements described in the previous section for probe type identification at a panel and per panel specialization of the information provided by a probe are handled.

5.1 Point Plot Panels

The first panel (shown in the left half of figure 7) is an example of a *point plot panel* used to generate a scatter plot. As an option, only points representing simulated activity over a limited past history from the most recent event time are kept for display. In this example, resource load[5] information is provided by the operator-load and evaluator-load component probes attached respectively to the operators and evaluators of the system.

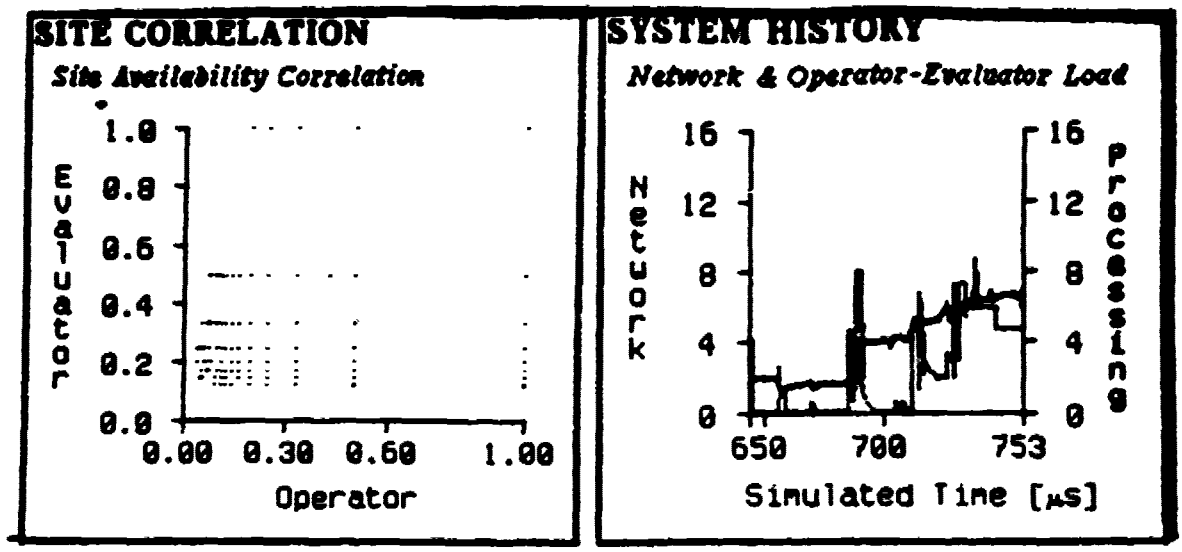


Figure 7: Point Plot and Scrolling Line Plot Panels

The balance between the "availability" of the evaluator and operator of each site, that is, the complements of their respective loads, is displayed during the simulation as events are processed that change this measure. In order to avoid capturing information at too fine a temporal granularity, previously gathered information for a given site is overwritten if it is within a given sampling interval of the new information. Information that is beyond a given history range is dropped. The scale of availabilities displayed is fixed between 0 and 1.0. The panel specification to declare all this and to also stipulate the axis labels is shown in figure 8.

[5]Resource load is defined as (1 - 1 / (1 + aggregate-queue-length)) where the aggregate queue-length is the sum of the lengths of all queues providing work for the resource.

```
'((("Operator") (0 1.0) (- 1 (:operator-load :busy)))      ;Bottom axis
  (("Evaluator") (0 1.0) ((- 1 (:evaluator-load :busy)))) ;Left axis
  :find (find-sample-distinct (:simulator :time) .sampling-interval)
  :show (recent-history (:simulator :time) .point-panel-history-range 0))
```

Figure 8: Site Correlation Panel Specification

5.2 Scrolling Line Plot Panels

An example of a *scrolling line plot panel* is shown in the right half of figure 7. This panel sums the loads seen by the resources in the simulated system and displays this as a strip chart, the "system history". Some of the same probe load information used by the previous panel is used in this panel as well, but with different transformations defined in the panel specification as shown in figure 9.

```
'((("Simulated Time [us]") (.history-range) (:simulator :time))  ;Bottom
  (("Network") (0 .sites) (:net-output-load :busy save-sum))     ;Left
  (("Processing") (0 .sites)                                     ;Right
     (average (:evaluator-load :busy save-sum)
              (:operator-load :busy save-sum)))
  :find (update-history (:simulator :time) .sampling-interval)
  :show (recent-history (:simulator :time) .history-range 0))
```

Figure 9: System History Panel Specification

Line plot panels may have two independently scaled vertical axes. For the system history panel shown, the sum of network loads as indicated by the net-output components of the system is plotted against the left axis and the sum of the processing loads provided by the current average of the sums of the operator and evaluator loads is plotted against the right axis. Event time is plotted on the horizontal axis. The `update-history` function uses the component pointer to find the information previously saved for that component and records the current event time as the `(:simulator :time)` so that it may be used to display information correctly on the horizontal axis. The current sums of the evaluator loads and the operator loads measured by the system are stored in a record for the given event time (or a prior event time within the specified sampling interval) by the calls to the `save-sum` function specified as part of the *save* operation.

5.3 Self Scaling Line Plot Panels

Figure 10 illustrates both the self scaling of displays and the use of a display analysis operation. For this self scaling line plot panel, two pieces of data are collected for each operator in the system: the load on the operator, shown on the right axis, and the latency of the information it has most recently received. This last item is provided by the operator latency probe in two parts: (1) the interval between the creation of the information and its receipt by the net-input feeding the operator and (2) the interval between such receipt and the operator taking action on it. There are thus two curves plotted on the left axis. The specification stipulates a list for the left axis display. The elements of this list are the "net delay" and the sum of this measure and the "operator delay" monitored by the operator latency probe. Since both delays are non-negative, their sum must be at least as large as either one taken alone: the two curves may be superimposed but can not cross. The difference between the two curves is the incremental delay added by the operator.

The panel specification for the operator-network panel is shown in figure 11. In addition to transformations shown previously, an analysis function is stipulated for the *send* operation of the panel. The information saved from each of the probes sending `:update` messages to the panel is to be sorted from the greatest to the least values of the associated sum of delays described above. This information is to be saved as the operator latency rank and used as such to determine the position on the horizontal axis that the delay and load information will be displayed.

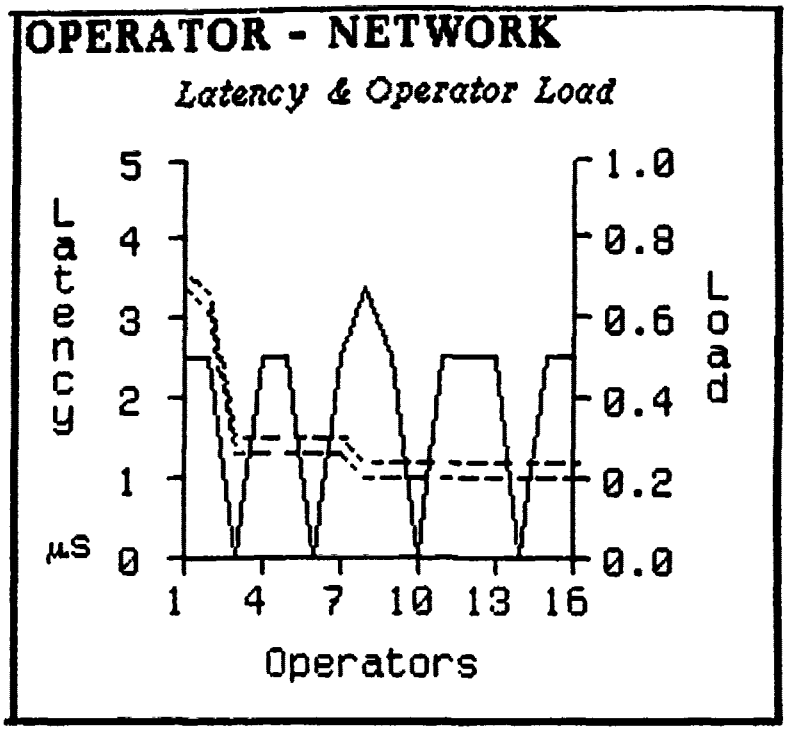


Figure 10: Self Scaling Line Plot Panel

```
'((("Operators") (1  .sites) (:operator-latency :rank))
  (("Latency" "us")) (0 nil) ;Second string: 90 degree baseline shift
   ((:operator-latency (:net-delay (+ :net-delay :operator-delay)))))
  (("Load") (0 1.0) (:operator-load :busy))
  :send (sort-arrays
          ((,#'> (:operator-latency (+ :net-delay :operator-delay))))
          ((:operator-latency :rank))))
```

Figure 11: Operator-Network Panel Specification

5.4 Boxes and Lines Panels

Perhaps the most intuitively satisfying of the types of panels available is the *boxes and lines panel*, a graphic representation of a circuit showing its components and their interconnections. An example of such a panel is shown the left part of figure 12. This class of panels uses information left behind by the structure editor when the circuit was defined. Its form is thus automatically generated. The position of the components ("boxes") and the connections between them ("lines") in the display are used to animate system operation. In the example shown, the shading (or color) of the boxes is used to indicate the availability of the *evaluators* in the simulated system as the simulation proceeds. Darkest shades indicate highest availability, that is, empty queues for utilization of the resource; lighter shades indicate lower availability, that is, longer queues. The lines between boxes indicate communication paths that are in use, that is, not ":free" at the time of the most recent *show* operation for the panel.

The panel specification for the *mapping panel*, an instance of a boxes and lines panel, is shown in figure 13. There are two specifications for the panel: one for the boxes and one for the lines. The specification for boxes in the panel stipulates that the availability of evaluators in the sites corresponding to the boxes displayed controls the shading of those boxes. The scale is defined to run from 0 to 1.0. The specification for lines in the panel uses the connection information reported for the net-output to determine line placement on the display. When the status is reported as :free, the connection information is dropped from the panel and the corresponding lines are removed.

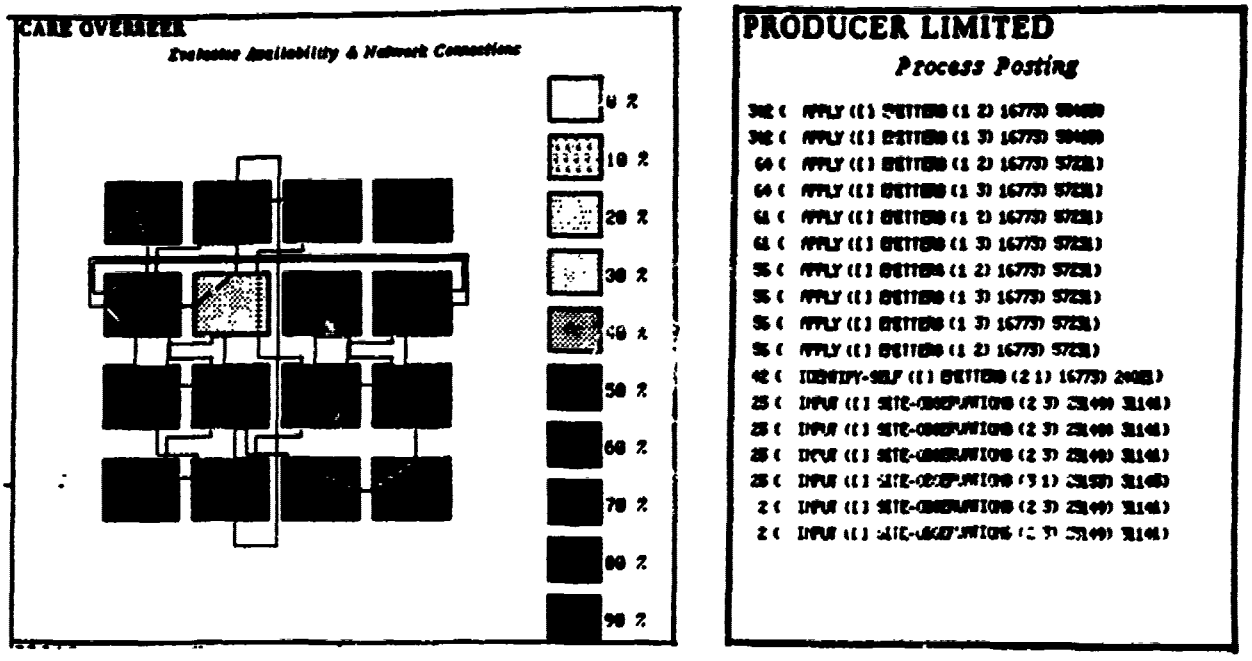


Figure 12: Boxes and Lines Panel and Scrolling Text Panel

```
'((("Evaluator Available") (0 1.0) (- 1 (:evaluator-load :busy))))
'(("Packet Trace") nil (:net-output-connection :points))
 (("Packet Status") nil (:net-output-connection :status))
  :find (find-and-remove .#'eq (:net-output-connection :status) :free)))
```

Figure 13: Mapping Panel Specification

5.5 Scrolling Text Panels

Sometimes, the most appropriate way to display information is to show it as text. Based on a similar facility provided by the underlying Lisp system, the *scrolling text panel* provides a scrollable window into lines of text. In the right part of figure 12, the delay in each process execution while waiting for something to do, that is, the event time interval spent waiting for an appropriate task to appear on a certain stream of tasks, is shown together with the process that finally produced the awaited work. This information is sorted so that the text lines appear from the greatest stream waiting interval to the least.

```
'((() ("~4D ~A")
   ((fix (:stream-waiting :interval))    ;first field
    (let* ((origins (packet-origin (:stream-waiting :packet)))
           (origin (if (listp origins) (first origins) origins)))
      (remote-address-local origin))))  ;second field
  :send (sort-arrays ((.#'> (:stream-waiting :interval))) nil))
```

Figure 14: Producer Limited Process Panel Specification

The values and formats used for display in a scrolling text panel are defined much as in previously defined panels. Format control strings take the place of scale information. As usual, values are described by a list of forms, each one of which specifies the transformations to perform on information received from probes. The example specification in figure 14 shows the generality with which probe information can be incorporated in Lisp expressions

to produce transformation specifications. The information used to generate the value for the second field of the text display is based on the origin of the task packet that arrived on the stream the process was waiting for.

5.6 Noting Simulation Parameters

The CARE component models are parameterized through menu interaction as shown in figure 15 to allow easy variation of their performance characteristics relative to each other. Additionally, the site model parameterizes alternative routing strategies: *directed*, that is, blocking when progress can not be made toward the goal; *spiraling* around the goal if progress toward it is blocked; and *dithering*, that is, routing away from the goal even if only the last link towards it remains to be acquired. The rate at which each site accepts application data is also a parameter, the *data rate* and can be used by an application to control how hard it drives the simulated system.

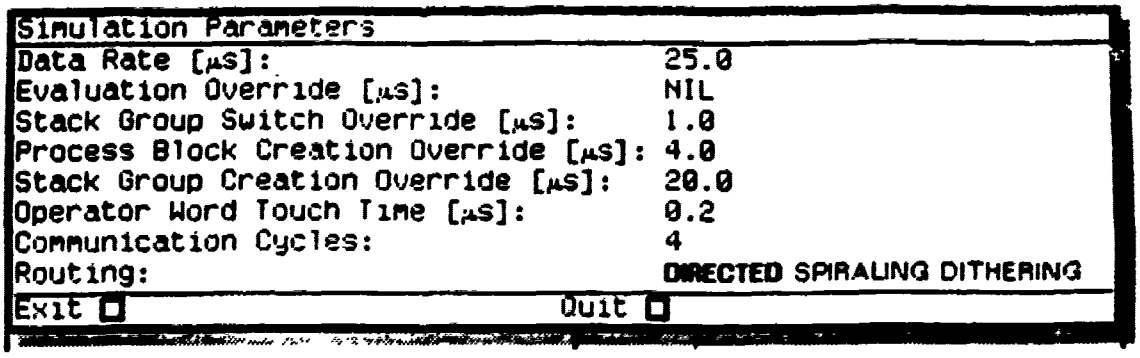


Figure 15: Parameter Menu

Many of the CARE parameters are specified as *overrides*. If not specified, the corresponding performance is taken as measured on the simulation machine. Thus, the *evaluation override*, that is, the time to perform an evaluation can be specified as non-nil in order to fix the time that each user evaluation will take. (This is useful in making runs repeatable for debugging). The time that it takes to switch context can be specified as the *stack group switch override*. Similarly, the time to create a process control block and a stack context for that process can be taken as given rather than measured by specifying respectively the *process block creation override* and the *stack group creation override*.

The time required for operator execution is modeled in terms of the number of words the operator must manipulate in handling a given message. The manipulation time per word is specified by the *operator word touch time*. Lastly, the performance of the communication subsystem is specified as *communication cycles*. This is done in terms of the minimum number of evaluator data path clock times (that is, event times) required for a 32-bit word to pass a given point in the network. Thus the parametric specification, "4 communication cycles", dictates that 8 bits may cross such a boundary each time the evaluator passes through one event time. If the communications path were narrower or the base communication clock rate were lower, a higher number would be specified.

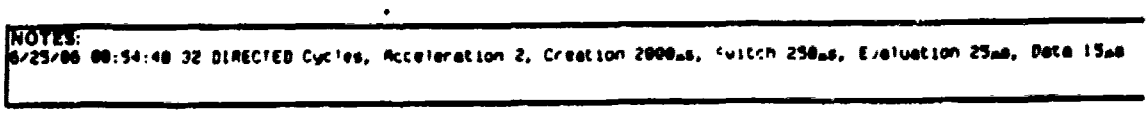


Figure 16: Annotation Panel

The last example of SIMPLE panels is the annotation panel as illustrated in figure 16. This

is used to (automatically) record the date, time, and parameters of the simulation run as well as any other information the user chooses to keyboard into it.

5.7 An Instrument Screen

All these panels are put together in an instrument screen according to a set of layout constraints manipulated by the underlying window system. The finished screen might look like figure 17. The instrument screen is redrawn at a rate set by the user. By experience, it is often better to update the screen at a frequency low enough to let the user interpret each screen comfortably than at the maximum rate possible. This approach also restricts the computing resources consumed by the instrumentation system. More focused approaches to controlling instrumentation load on the system include the ability to freeze selected panels and disconnect selected probes during a simulation run.

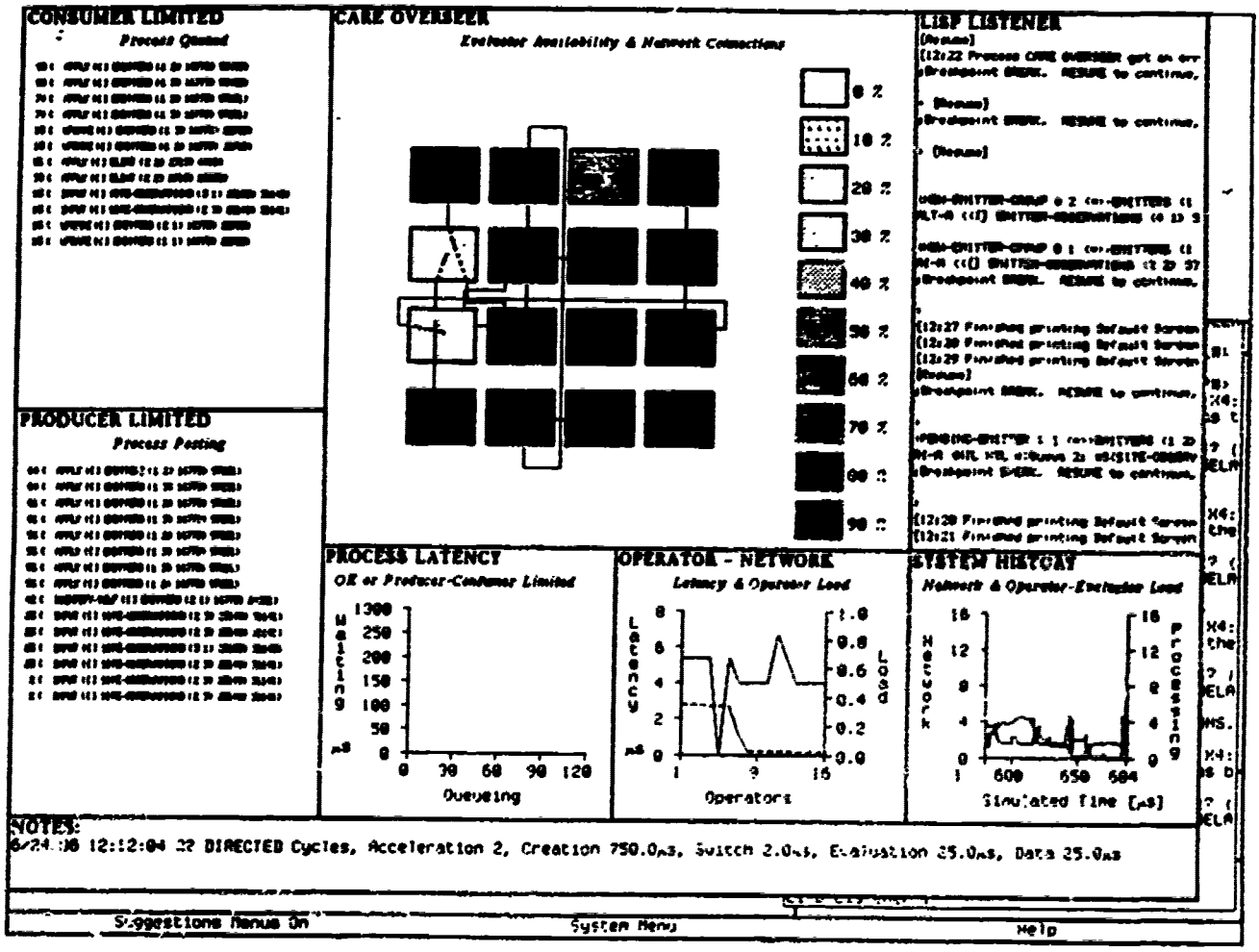


Figure 17: Overseer Instrument

6 USING PROGRAM DEVELOPMENT TOOLS

The SIMPLE/CARE simulation system is integrated into the underlying Lisp machine program development environment. The objects and data structures at both the component model and application language interface have abstraction interfaces that provide summary

state information when they are displayed in text form. These text abstractions are "mouse sensitive" in the development machine environment and so can be inspected at successively finer levels of detail as desired.

In figure 18, the net-output components of the site at grid coordinates (3 2), the particulars of the net-output on the east side of the site (that is, net-output-3), and a summary of all the sub-components of the site at (3 2) are being inspected. This same kind of view into the progress of a simulation is provided in the debugging process and may, as shown in figure 19, refer to the conceptual entities of the application that is driving the simulated system.

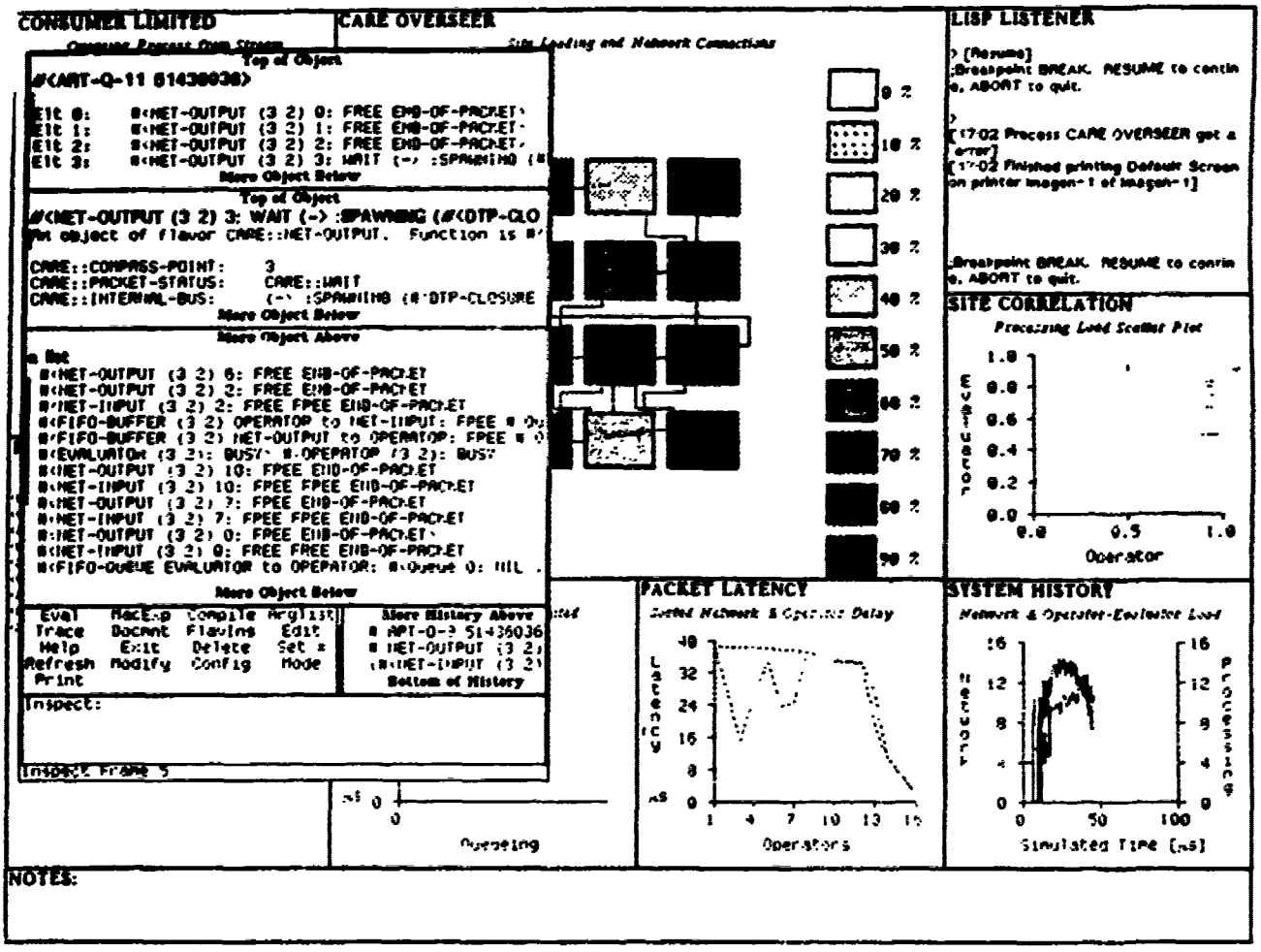


Figure 18: Inspecting Simulated Components

In the example shown in figure 19, a `distributer` process running on the evaluator at site (1 1) has made an improper call on the `update-locale` function during execution of its `:start` method. It might have been appropriate to investigate this situation in terms of the modeled components. That could be done, for example, using the debugger to inspect the evaluator component, its enclosing site, related net-output components, or whatever else at the component model level seemed relevant. In this case, what was done was to use a few mouse clicks to indicate interest in the source file for the `distributer :start` method generating the problem. It was brought up for review and control was then transferred to an editor using the underlying program development environment as shown in figure 20.

Because of the implementation system chosen for the realization of SIMPLE/CARE, at any point in the simulation, procedures either in the application or in the component models can be modified, incrementally recompiled (within a few seconds), and be made effective for all

calls on them -- even those in the interrupted stack frame. Thus simulation execution can be backed up to some previous point in the stack frame and retried (given that intermediate side effecting code, if any, is safely re-executable).

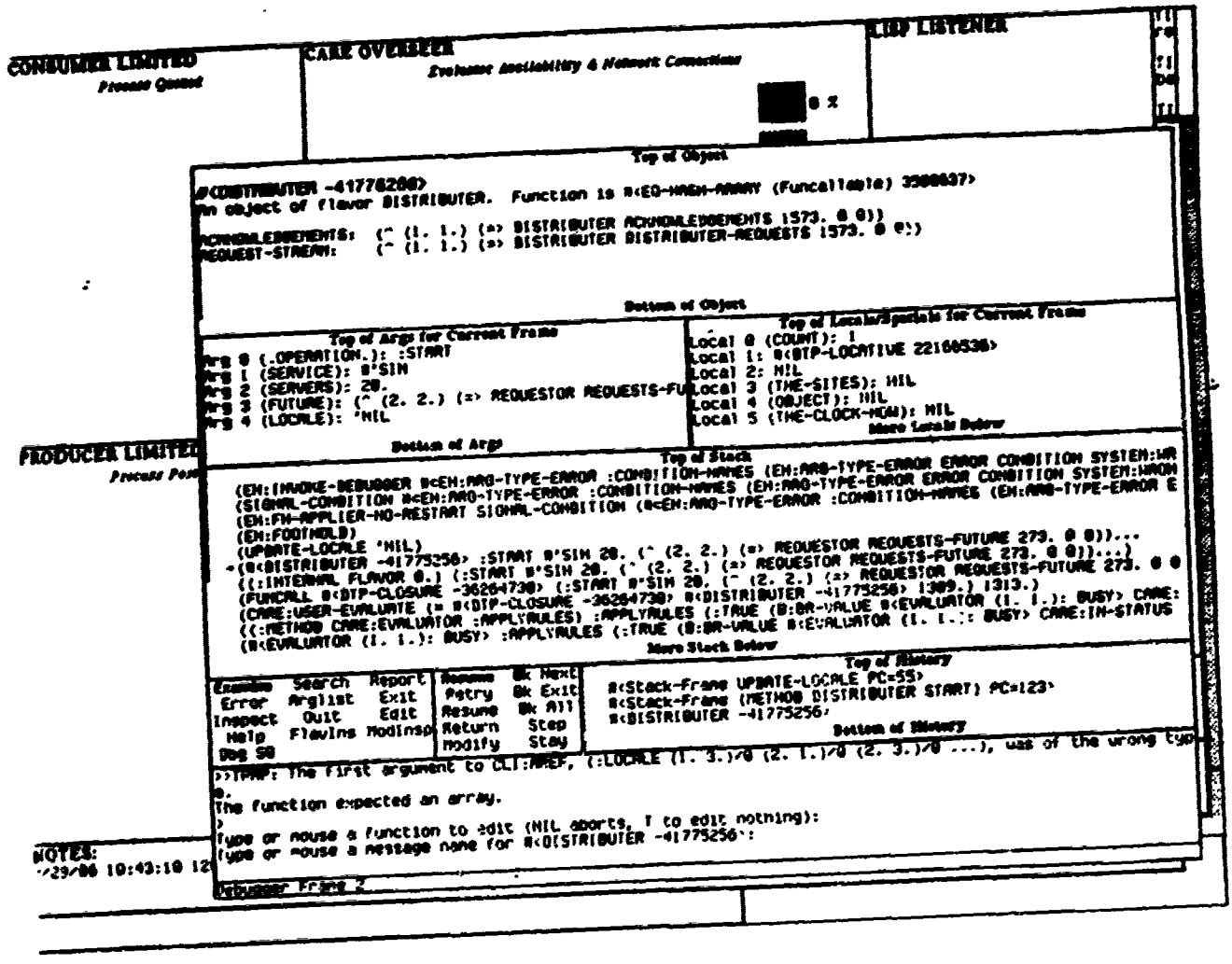


Figure 19: Debugging A Simulation

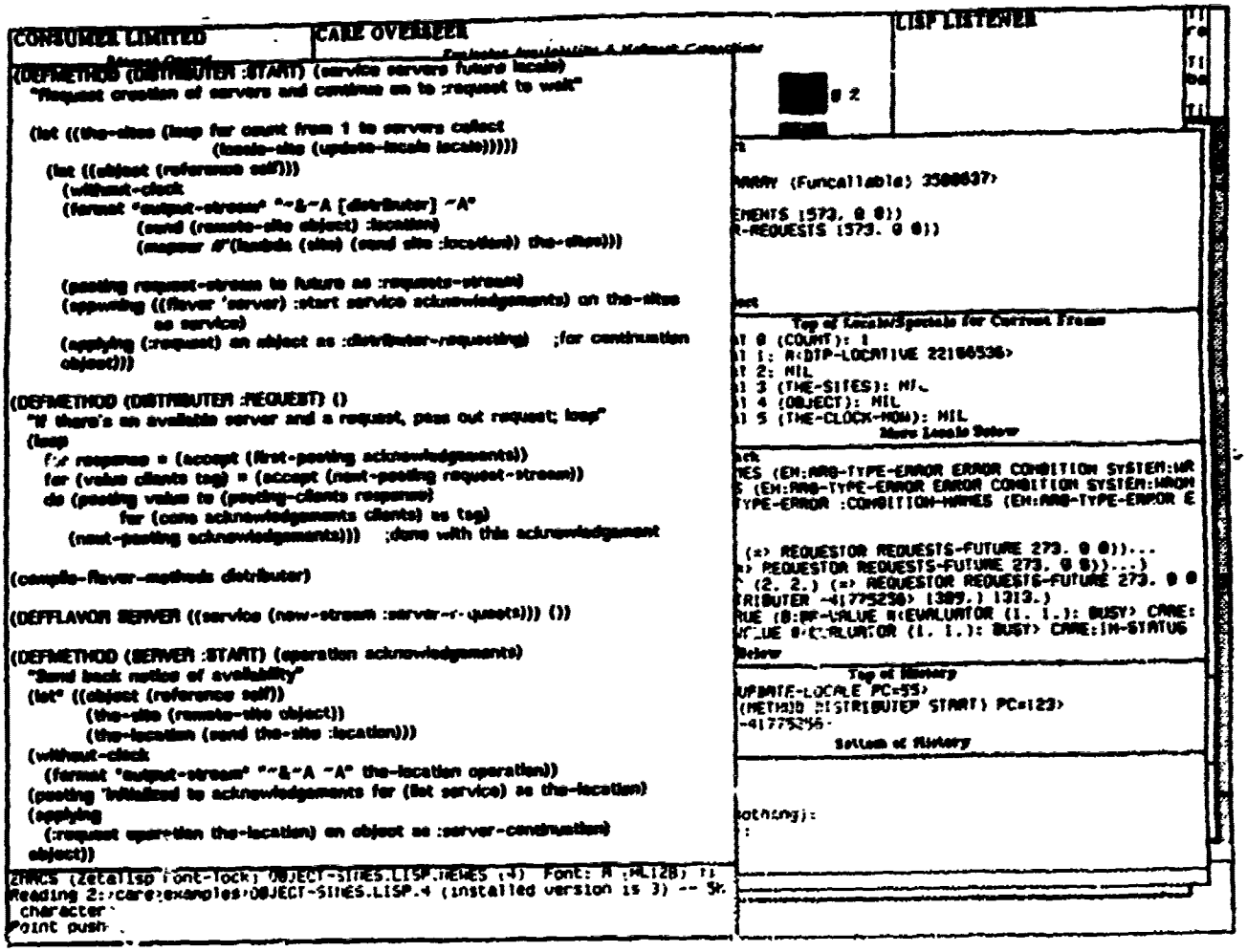


Figure 20: Changing Application Code

7 CONCLUSIONS

The goals of simulation flexibility and simulation environment completeness have been dealt with in the ways described throughout this paper. In summary, the system is flexible in that it supports:

- Arbitrary data types and lengths in simulation. The information whose flow and creation is controlled by simulated components may be of arbitrary complexity -- from numbers and keywords to procedure bodies and execution environments.
- Instantaneous effect of definition change at both the application and component modeling level (even during a simulation run).
- A broad range of instrumentation customization. Customizations may involve arbitrary expressions for probe data transformations, many to many probe to panel mappings, information from summary analyses on one panel's data included in another, and control of what state is saved and for how long.
- Separation of probe and component definitions to facilitate their independent modification.
- An application language interface that is easily extended or changed without recasting the information flow control described by the component behaviors.

While there is always room for additional capability[6], SIMPLE/CARE is a usefully complete system. It now includes:

- Supplied components for a network multiprocessor simulation with many of their parameters customizable by menu interactions.
- A hierarchical structure editor that currently provides automatic grid and torus composition operators. (Automated composition of richer topologies, such as hypercubes, has been provided for in the basic design).
- A rule language that supports a synchronous design style without incurring the overhead of (naive) synchronous simulation.
- Method invocation for functional simulation that is integrated into the behavioral simulation rule system and which provides for operations by and on both local and hierarchically related components.
- Method specification design aids provided by the underlying program development environment (for example, method dictionaries and quick access to method sources from the debugging system).
- An evolved set of panel templates providing sorted, scrollable text lines as well as self and fixed scaling, "two and a half" dimensioned, history sensitive displays which may be scatter plots, strip charts, line graphs, intensity maps, and signal animations.

We set off to build a multiprocessor simulation system with performance adequate for the understanding of multiprocessor systems executing significant applications. The SIMPLE/CARE simulation system has been used to study the operation of "expert systems" of respectable size [2]. Depending on instrumentation load, these studies have involved simulation runs from 20 minutes to several hours each. While faster would surely be better, performance has proven adequate to these needs.

[6] A *histogram panel*, for example, is just now being added to the system

8 ACKNOWLEDGEMENTS

This work stands on the shoulders of its predecessor, the Palladio system, designed and implemented by Harold Brown and Gordon Foyster. Our functional goals were more restrictive than theirs so we had the luxury of design by simplification. Without their implementation base, it would have been hard to know even where to begin.

Many hands and minds have contributed to the development of SIMPLE/CARE. We are particularly indebted to the work of Russ Nakano who started off to do a simple learning exercise and ended up doing a particularly careful modeling of a intricate signalling protocol.

References

1. Brown, Harold, Christopher Tong, and Gordon Foyster. "PALLADIO: An Exploratory Design Environment for Integrated Circuits." *IEEE Computer 16* (December 1983).

2. Harold D. Brown, Eric Schoen, and Bruce A. Delagi. An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures. Tech. Rept. STAN-CS-86-1136 or KSL-86-69, Stanford University, October, 1986.

3. Greg Byrd, Russell Nakano, and Bruce Delagi. A Point-to-Point Multicast Communications Protocol. Tech. Rept. KSL-87-02, Knowledge Systems Laboratory, Stanford University, January, 1987.

4. Daniel Weinreb and David Moon. *Lisp Machine Manual*. Symbolics, Cambridge, MA, 1981.

Knowledge Systems Laboratory
Report No. KSL 87-65

November 1987

Instrumented Architectural Simulation

by

Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura,

and Greg Byrd

Digital Equipment Corporation
Maynard, Massachusetts 01754

Stanford University
Stanford, California 94305

This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Greg Byrd was supported by an NSF Graduate Fellowship and by the Stanford University, Department of Electrical Engineering.

Instrumented Architectural Simulation

Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd

Digital Equipment Corporation
Maynard, Massachusetts 01754

Stanford University
Stanford, California 94305

ABSTRACT

Simulation of systems at an architectural level can offer an effective way to study critical design choices if (1) the performance of the simulator is adequate to examine designs executing significant code bodies -- not just toy problems or small application fragments, (2) the details of the simulation include the critical details of the design, (3) the view of the design presented by the simulator instrumentation leads to useful insights on the problems with the design, and (4) there is enough flexibility in the simulation system so that the asking of unplanned questions is not suppressed by the weight of the mechanics involved in making changes either in the design or its measurement. A simulation system with these goals is described together with the approach to its implementation. Its application to the study of a particular class of multiprocessor hardware system architectures is illustrated.

1 INTRODUCTION

Simulation systems are quite often developed in the context of a particular problem. To a degree, this is true for SIMPLE, an event based simulation system, and CARE, the computer array emulator that runs on SIMPLE.[1] The problem motivating the development of both SIMPLE and CARE was the performance study of 100 to 1000-element multiprocessor systems executing a set of signal interpretation applications implemented as "1000 rule equivalent expert systems" [2].

A set of constraints pertinent to this problem governed the design of SIMPLE/CARE. The applications represented significant bodies of code and so simulation run times were expected to be an important consideration. Moreover, the issues involved with the interactions of multiprocessor system elements were sufficiently unexplored prior to simulation that simplifications in the CARE system model, specifically with respect to element interactions, were suspect. This need for detail was, of course, in tension with the need for simulation performance. The ways that simulated system components would be composed into complete systems was initially difficult to bound. Further, it was clear that the models of these components would be elaborated over time and would undergo substantial change as design concepts evolved. It was also clear that the ways of examining the operation of these components would change independently (and at a great rate) as early experience indicated what alternative aspect of system operation *should* have been monitored in any given completed run.

The design goals that emerged then were (1) that the simulation system should support the management of substantial flexibility with regard to simulated system structure, function, and instrumentation and (2) that, in order to accomplish runs in acceptable elapsed times, the detail of simulation should be particularly focused on the communications, process scheduling, and context switching support facilities of the simulated system -- that is, on just those aspects of system execution critical to multiprocessor (as opposed to uniprocessor) operation.

[1] SIMPLE and CARE were developed by the authors at the Knowledge Systems Lab of Stanford University. SIMPLE is a descendent of PALLADIO [1] optimized for the subset of PALLADIO's capabilities relevant to hierarchical design capture and simulation. It is written in Zetalisp [3] and currently runs on Symbolics 3600 machines and TI Explorers.

This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Greg Byrd was supported by an NSF Graduate Fellowship and by the Stanford University Department of Electrical Engineering.

1.1 Design Time Interaction And Run Time Operation

Encapsulation of the state of design components with the procedures that manipulate that state is one clear way to manage design evolution. Such encapsulation partitions the design along well defined boundaries. Components (by and large) interact with other components only through their *ports*. Connections between components terminate at such ports. When a system simulation is initialized, connections are traced so that for every port, the simulator knows the connected (terminating) ports together with their containing components. Once such initialization is complete, that is, throughout the simulation run, assertions about the state of a port of one component can be directly translated to assertions about the state of connected ports of other components.

Partitioning issues of **system structure**, **component behavior**, and **instrumentation** into separate domains of consideration helps in managing a design that is both fluid and complex. **System structure**, that is, the relationship between components, can be specified through use of an interactive, graphics structure editor and is largely independent of component function per se. **Component behavior** is encapsulated in a set of definitions pertinent to the given class of component. Each component in a SIMPLE simulated system is a member of a class defined for that component type. **Instrumentation** is automatically and invisibly made part of the definition of each simulated component that is to be monitored during a run. This is done by arranging that the class of every component to be monitored is a specialization of the general *instrumented-box* class. The basic data structures and procedures for monitoring simulated components and maintaining the organizational relationships between each component and its related instrumentation are inherited through this general, ancestral class and are thus made separate, substantially independent consideration in the design.

A further partitioning of concerns is employed to separate out the definition of the application programming language interface and its support (as provided by CARE) from the underlying information flow control governing component behavior. The behavioral descriptions of components (which are expressed as sets of condition/action rules) deal generically with gating information, independently of the structure of the information, between ports of the component and its internal state variables. This is separated in the component model definitions from the functions performed to create and manipulate the information so gated. The simulated implementation of the application programming language support facilities, on the other hand, relies only on the specifics of the information and its structure and plays no part in gating it between the components of the system. Changing the definition of the application language is thus done independently of changing component flow control behavior. The application programmer and the implementer of the application language interface may use whatever data structures seem suitable to them, be they numbers and keywords or procedure bodies and execution environments. The simulation system doesn't care.

The *component probe* definitions, that is, the specifications of what information should be captured for each component type, are separated from the descriptions of the behavior of such components. In designing for flexibility in the instrumentation system, it turned out to be important to further divide the information presentation from the information collection issues. The mapping from particular component probes to particular *instrument panels* and the transformations to be applied to the information as it passed from a given kind of probe to a given panel (and between panels) is captured in the *instrument specification.* This is a definition of what kinds of panels are included in an *instrument*, how they fit on an *instrument screen*, how they are labeled and scaled, and what information from which kinds of probes are displayed on each panel. The instrument specification also indicates what kinds of probes are to be connected to which kinds (that is, which classes) of components in the system.

Putting together all the definitions of components, component probes, panels, instruments, applications interfaces, and inter-component relationships is done in a set of design time interactions by a system architect. These interactions are used by the simulation system to generate efficient run time representations so that simulation performance goals can be met. Figure 1 illustrates the partition between design time interactions and simulation run time operation. Structure editing pulls together components from the component library to produce a *circuit*. Associated with some components in the library, there are definitions for the syntax and underlying mechanisms of a multiprocessor applications language. These specify the interface used to provide the program input to the multiprocessor system being simulated.[2]

[2]The language primitives supplied can be used to define multiprocessor language interfaces for either shared-variable or value-passing paradigms. As supplied, the language interface built on these primitives supports value-passing on streams between objects but alternative interfaces can be (and have been) easily defined in terms of the given primitives.

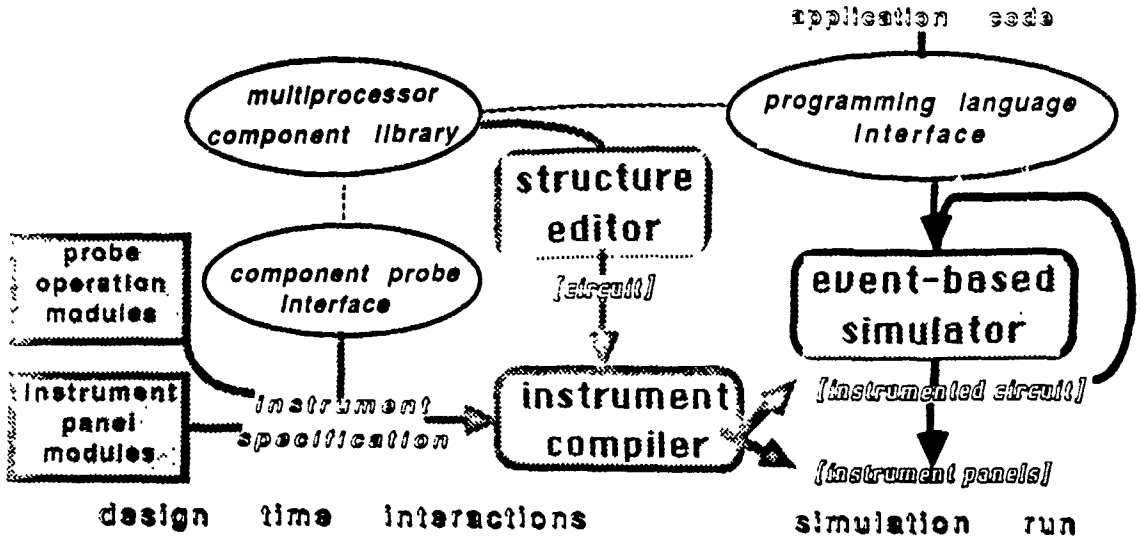


Figure 1: Design Time Interactions and Run Time Representations

The definitions used to generate component probes are associated with each library component to be monitored. There may be several such definitions, each appropriate to measuring a different aspect of the associated component's operation. An instrument specification selects from these definitions, elaborates them with selections from a set of *probe operation modules* to include any pre-processing (for example, a moving average) to be calculated by the probe, and indicates under what conditions what information from the probe is to be sent to which panels of the instrument and how it is to be transformed and displayed there. Instrument specifications also partition the screen among the panels of the instrument. The end product of these design time interactions is an *instrumented circuit* and an *instrument*. The instrument comprises a set of instrument panels and a set of constraints relating them to the instrument screen. The instrumented circuit ties together instances of components, probes, and panels for a simulation run.

For each defined class of component and its associated probes, the design time interactions produce code bodies that accomplish simulation operations during a run. It is an attribute of the underlying Lisp base of the simulation system that changes in these definitions have immediate effect even during a simulation run -- an important capability during debugging.

2 STRUCTURE AND COMPOSITION

Design time interactions to specify a system include the establishment of component relationships. Such specifications can be said to accomplish the composition of the system from its components and so define its structure. SIMPLE supports hierarchical composition: components may be described in terms of a fixed set of relationships among their sub-components. Additionally, such composite components may have function beyond what can be inferred strictly from their composition. All this can then be included a higher level composite and so on indefinitely until the top level "circuit", the system structure, is reached.

Composition is described graphically and interactively in SIMPLE by picking a previously specified component type from a menu, placing it in relationship to other components with "mouse" movements, and, through the same means, specifying the connections between its selected ports and those of other components.

Although any connection of components can be created by the means noted previously, for some repetitive, well patterned systems of connections, composition can be automated. The CARE library includes a component, the *iterated-cell*, which represents a template for the creation of composite components by iteration of a unit cell. The specializations include a method for responding to a request to provide a wiring list. Such a list associates each source port of a cell with the corresponding destination port (in terms of port names) and the position of the destination cell relative to the source cell in the iterated structure. The iterated cell component uses this information to make the required connections between each of its constituent cells.

3 INSTRUMENTATION

The results of a simulation are primarily the insights it provides into the operation of the simulated system. The "insight" we frequently experienced using an early version of the simulation system was that more interesting results could have been produced by the run just completed if only the instrumentation had been different. With this in mind, the design for the current version of the simulation instrumentation system was aimed at flexibility. This was attained without significant performance impact by building efficient run-time system structures before each run, as outlined in section 1.1, from the declarations defining the instrumentation.

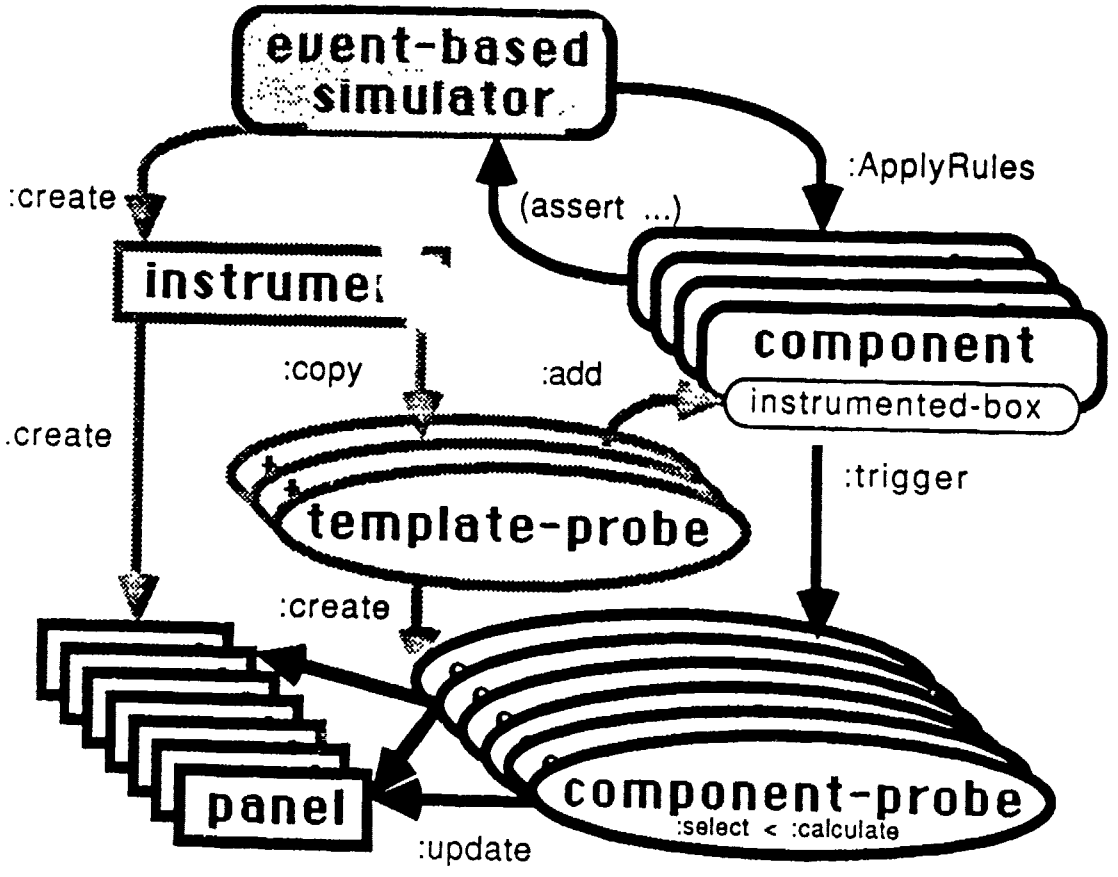


Figure 2: Instrument System Organization

The organization of the instrumentation system is pictured in figure 2. The simulator interacts with component instances through assertions, that is, calls on an `assert` function, in behavior rules (the methods associated with `:ApplyRules` messages). All instrumented components are specializations of an *instrumented-box* (as well as other classes). After each

invocation of :ApplyRules for such components, the :ApplyRules method for a generic instrumented-box is applied. This causes invocation of the :trigger method for each *component-probe* associated with that component. Data from component probes is collected and displayed by *instrument panels*. Since this flow of measurements is accomplished by means invisible to the the writer of behavior methods for a component, the concerns surrounding component design are effectively partitioned from component instrumentation. Panels are put together in an instrument screen according to a set of layout constraints manipulated by the underlying window system. The finished screen might look like figure 3.

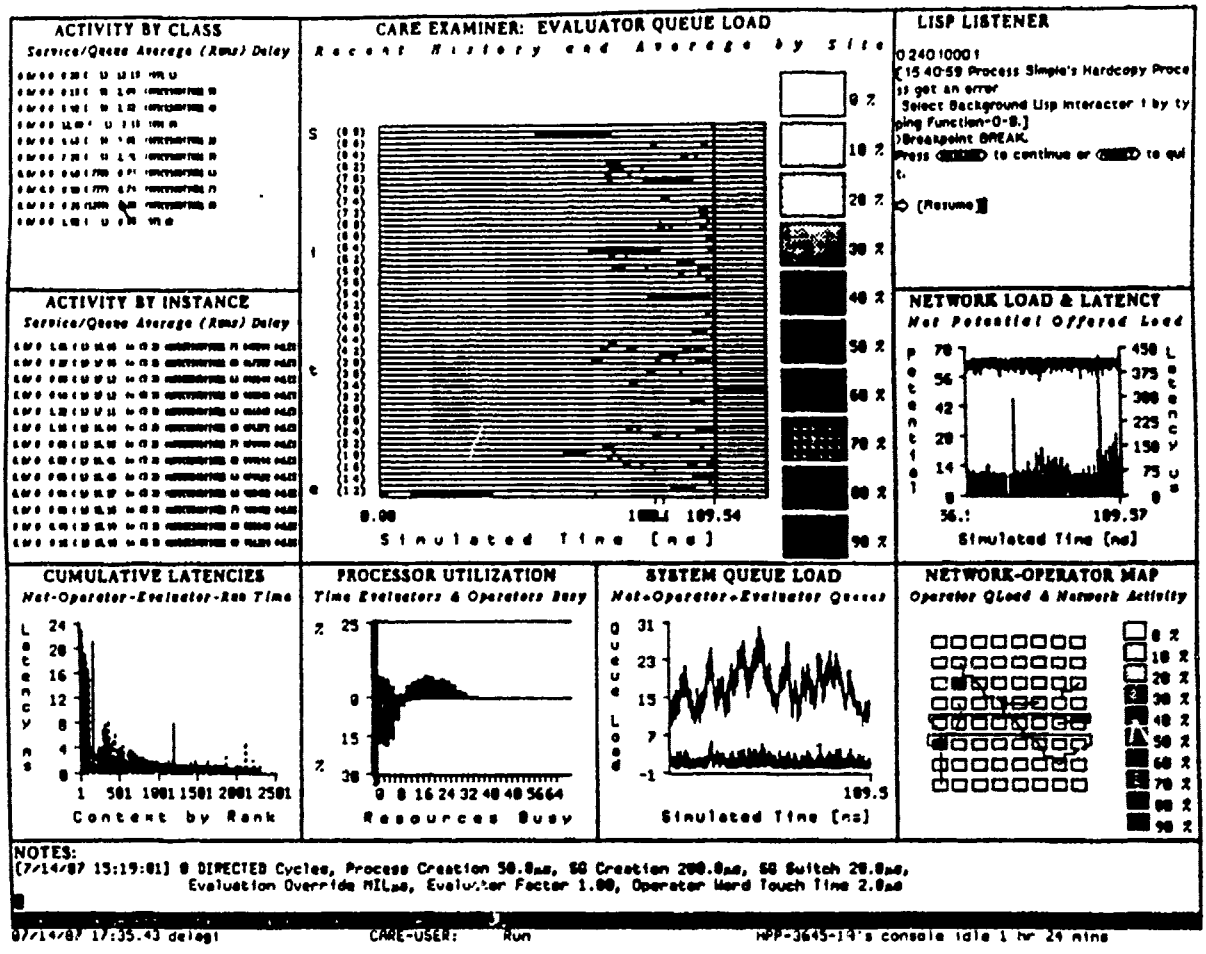


Figure 3: Overseer Instrument

4 CONCLUSIONS

The design goals of simulation flexibility and simulation environment completeness have been supported as discussed above. In summary, the system is flexible in that it supports:

- Arbitrary data types and lengths in simulation. The information whose flow and creation is controlled by simulated components may be of arbitrary comp'exity -- from numbers and keywords to procedure bodies and execution environments.
- Instantaneous effect of definition change at both the application and component modeling level (even during a simulation run).
- A broad range of instrumentation customization. Customizations may involve arbitrary expressions for probe data transformations, many to many probe to panel mappings, information from summary analyses on one panel's data included in another, and control of what state is saved and for how long.

- Separation of probe and component definitions to facilitate their independent modification.
- An application language interface that is easily extended or changed without recasting the information flow control described by the component behaviors.

While there is always room for additional capability, SIMPLE/CARE is a usefully complete system. It now includes:

- Supplied components for a network multiprocessor simulation with many of their parameters customizable by menu interactions.
- A hierarchical structure editor that currently provides automatic grid and torus composition operators. (Automated composition of richer topologies, such as hypercubes, has been provided for in the basic design).
- A rule language that supports a synchronous design style without incurring the overhead of (naive) synchronous simulation.
- Method invocation for functional simulation that is integrated into the behavioral simulation rule system and which provides for operations by and on both local and hierarchically related components.
- Method specification design aids provided by the underlying program development environment (for example, method dictionaries and quick access to method sources from the debugging system).
- An evolved set of panel templates providing histograms and sorted, scrollable text lines as well as self and fixed scaling, "two and a half" dimensioned, history sensitive displays which may be scatter plots, strip charts, line graphs, intensity maps, and signal animations.

We set off to build a multiprocessor simulation system with performance adequate for the understanding of multiprocessor systems executing significant applications. The SIMPLE/CARE simulation system has been used to study the operation of "expert systems" of respectable size [2]. Depending on instrumentation load, these studies have involved simulation runs from 20 minutes to several hours each. While faster would surely be better, performance has proven adequate to these needs.

5 ACKNOWLEDGEMENTS

This work stands on the shoulders of its predecessor, the Palladio system, designed and implemented by Harold Brown and Gordon Foyster. Our functional goals were more restrictive than theirs so we had the luxury of design by simplification. Without their implementation base, it would have been hard to know even where to begin.

Many hands and minds have contributed to the development of SIMPLE/CARE. We are particularly indebted to the work of Russ Nakano who started off to do a simple learning exercise and ended up doing a particularly careful modeling of a intricate signalling protocol.

References

1. Brown, Harold, Christopher Tong, and Gor n Foyster. "PALLADIO: An Exploratory Design Environment for Integrated Circuits." *IEEE Computer 16* (December 1983).

2. Harold D. Brown, Eric Schoen, and Bruce A. Delagi. An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures. Tech. Rept. STAN-CS-86-1136 or KSL-86-69, Stanford University, October, 1986.

3. Daniel Weinreb and David Moon. *Lisp Machine Manual*. Symbolics, Cambridge, MA, 1981.

Knowledge Systems Laboratory
Report No. KSL 88-53

September 1990

Care User Manual

Version 0 (for Release 0)

by

Bruce A. Delagi, Nakul Saraiya, Greg Byrd, Sayuri Nishimura

KNOWLEDGE SYSTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, California 94305

CARE User Manual

Version 0 (for Release 0) [1]

Bruce A. Delagi Nakul P. Saraiya Gregory T. Byrd
Sayuri Nishimura

KNOWLEDGE SYSTEMS LABORATORY
Stanford University
Stanford, CA 94305

and

DIGITAL EQUIPMENT CORPORATION
Palo Alto, CA 94301

July 30, 1990

[1]This work was supported by DARPA Contract F30602-85-C-0012, by NASA Ames Contract NCC 2-220-S1, by Boeing Contract W266375, and by Digital Equipment Corporation.

Chapter 1

An Overview of SIMPLE/CARE

This chapter corresponds to the forthcoming Release 1 of SIMPLE/CARE. *However, it is provided in this version of the manual (corresponding to Release 0) since the basic architecture of* SIMPLE/CARE *has remained substantially unchanged between releases 0 and 1, even though many of the interfaces to* SIMPLE/CARE *have changed. The chapter should therefore be read to simply gain an understanding of how the system works; later chapters in this manual will clarify interface details.*

1.1 Introduction and Overview

Simulation systems are often developed in the context of a particular problem. To a degree, this is true for SIMPLE, a general-purpose modelling system, and CARE, the multiprocessor architecture simulator that runs on SIMPLE.[1] The problem motivating the development of SIMPLE/CARE was the performance study of hundred- to thousand-element multiprocessor systems executing a set of signal interpretation applications that were to be implemented in several alternative programming formalisms.

This problem offered a set of constraints that governed the design of SIMPLE/CARE.

- The kinds of multiprocessor system components that would be needed and the ways in which these would be composed into complete systems was initially difficult to bound. This meant that component models would be modified and elaborated over time as design concepts evolved.
- It was evident that instrumentation requirements were similarly fluid. Results from early simulation runs were likely to identify alternative aspects of system operation that *should* have been monitored, but were not. Further, since the simulator was to be used by system architects and applications

[1] SIMPLE is a descendent of the PALLADIO VLSI design system [1], that has been optimized for the subset of PALLADIO's capabilities relevant to hierarchical design capture and simulation. SIMPLE was originally developed using Zetalisp; it currently uses Common Lisp with Flavors.

programmers alike, their individual needs for detail in the view of system operation had to be satisfied. It was thus important that instrumentation could be varied both rapidly and independently of the system models.

- The applications represented significant bodies of code, so simulation run times had to be minimized. This meant that some simplifications in system models were indicated. On the other hand, the interactions of multiprocessor system elements were the least understood aspect of system operation. Thus, it was desirable that the system models capture the details of these interactions; otherwise, simulation results would be suspect.

The primary design goals that emerged then were:

1. that SIMPLE should offer significant flexibility with regard to the specification of system models and their instrumentation, while maintaining efficiency in simulating these models; and,
2. that, in order to accomplish runs with acceptable elapsed times, CARE should particularly focus on the details of a multiprocessor system's communications and scheduling support facilities: aspects of system execution critical to multiprocessor (as opposed to uniprocessor) operation.

The remainder of this section describes how the organizations of SIMPLE and CARE contribute towards meeting these goals.

1.1.1 The Organization of SIMPLE

SIMPLE provides flexibility[2] in specifying system models by partitioning issues of *system functionality* and *system instrumentation* into separate, largely independent domains of consideration. In both areas, SIMPLE further partitions concerns as shown in figure 1.1 and described below.

System Function

The principal abstraction supported by SIMPLE to specify system models is the *component*. A component represents a fragment of system functionality by encapsulating some private state along with the procedures that define how that state changes over time. A component is therefore naturally represented by an object. The component abstraction partitions the design along well defined boundaries since, by and large, components interact only through their defined *ports*. Connections between components terminate at such ports so that, during a simulation run, assertions about the state of a port of one component can be directly translated to assertions about the state of a connected port of some other component.

System structure defines how components are combined to form a larger system. This is specified incrementally: as definitions for each component type which describe the subcomponents (if any) a component

[2]Much of SIMPLE/CARL's flexibility and power derives from its Common Lisp implementation environment, which includes tools such as "on-line" inspectors and debuggers, and also provides a powerful object-oriented programming system, Flavors, with extensive capabilities for multiple inheritance and method combination. Flavors permits the evolutionary development of software 'libraries' for all aspects of the system design without sacrificing performance.

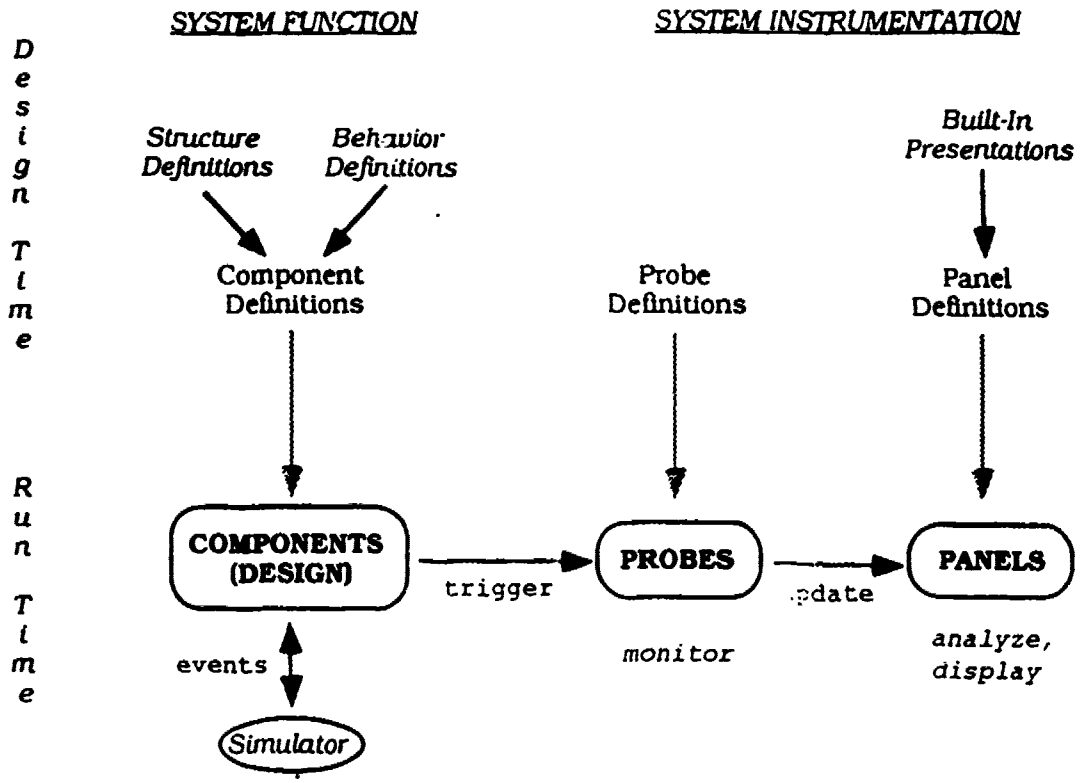


Figure 1.1: Simulator Organization

of that type contains and how their ports are to be interconnected. Optional definitions for geometric layout and routing allow the designer to view the structure graphically. These specifications are captured as procedures, allowing efficient, parameterized and programmable structure generation. A complete *design* is 'constructed' before a simulation run by the recursive generation of its parts, yielding a hierarchical network of interconnected components.

Component behavior defines how the state of a component changes over time. Behavior definitions are encapsulated as procedures relevant to each class of component, and can thus be developed mostly in isolation so long as interfaces are maintained. Behavior code is responsible for handling *events*—time-tagged state changes to a component's ports and internal state variables during a simulation run—in order to generate the local state changes 'caused' as a consequence. SIMPLE provides constructs that allow behavior code to be stylized as condition-action 'rules' to ease readability.

System Instrumentation

Every component automatically includes support for instrumentation because every component inherits the basic functionality required for monitoring it and for maintaining its organizational relationships with the instrumentation system. Ths allows instrumentation to be introduced into the design *non-intrusively*: without changing model function, and *incrementally*: as interesting aspects of a component's operation are identified.

SIMPLE factors system instrumentation into the details of *data capture*, *data analysis*, and *presentation*. This allows for the flexible intermixing of different capabilities for each of these concerns.

Component probe definitions specify what data should be captured for each component type. There may be several probe types for a component type, each appropriate to measuring a different aspect of the component's operation. Probes may make use of predefined modules to accomplish certain types of calculations (for example, moving averages) on captured data.

Panels bring together the data analysis and presentation aspects of SIMPLE's instrumentation system. They specify how the data supplied by probes is to be transformed through analysis, and how the results are to be displayed. SIMPLE has a basic library of *presentations*. class definitions which represent particular display styles such as histograms, intensity maps and scrolling line plots. It also provides a number of procedures to accomplish standard data analysis operations.

A panel is defined by customizing the appropriate presentation class with descriptions affecting its graphical appearance (e.g., legends, axis labels and scales), along with *interface specifications*. expressions using an augmented Lisp syntax to describe probe types, data transforms, and displayed quantities. Defined panels may then be aggregated into an *instrument*. which associates a named type and a screen layout policy with the collection of panels.

Instrumentation is 'attached' to a design before a simulation run by simply instantiating the appropriate instrument type with the design as a parameter. This results in the appropriate panels being created, at which time their corresponding interface specifications are compiled into efficient data structures and code that will accomplish the panel analysis and transformation operations. The required probes are also attached to components at that time. The end result is an *instrumented design* that ties together instances

of components, probes and panels for the simulation run.

1.1.2 The Organization of CARE

At the base level, CARE provides a library of multiprocessor components such as network interfaces, busses, processors, message coprocessors and memory controllers. These can be composed into a number of standard system configurations, such as toroidal networks or systems of hierarchical busses. Most components are parameterized, allowing variation in performance characteristics such as cycle times and channel widths, as well as choices on other aspects of system behavior, such as routing algorithms.

To satisfy the need for detail required in modelling multiprocessor system element interactions, the definition of network components is fine enough to capture each of the many operations that accomplish cut-through message routing of a packet of data in a torus network. To satisfy the runtime requirements of simulating complete applications, the processor models are coarse enough (and thereby fast enough) to ignore the details of simple processor operations that affect system operation only through their timing. Instead, this timing information is captured during the simulated execution of concurrent programs by dynamically running purely sequential segments of application code on the underlying machine and measuring their execution time.

Concurrent Programming Models

CARE defines parallel programming language extensions (collectively called LAMINA) for message passing, shared variable and functional programming models. The primitive mechanisms that support these language models are encapsulated within component definitions, but are decoupled from the underlying information flow control governing component behavior.

Component flow control actions deal generically with gating information between local ports and state variables, that is, communicating information, independent of its content. Language support actions, on the other hand, create and manipulate information based solely on its content, and play no part in communicating it between components. This separation of functionality allows the study of alternative communication protocols or topologies without modification to language interfaces and applications. Further, new language interfaces may be defined or existing ones changed without redefining the communications protocol used by the system components.

Instrumentation

CARE supplies a library of probe, panel and instrument definitions corresponding to particular multiprocessor systems and language models. For example, one CARE system architecture is a message-passing multicomputer that executes application programs using the concurrent object-oriented LAMINA extensions. An instrument for this system has probes which monitor the critical operations performed on messages both by application objects and by the resources of the underlying multiprocessor. These drive panels that display loads and latencies at the 'hardware' as well as application levels.

1.1.3 Using SIMPLE/CARE

A system architect develops multiprocessor component models and instrumentation through a set of design time interactions with the simulation system. The application developer, in turn, writes parallel code using the LAMINA language extensions or higher level frameworks derived from these. All the definitions—for models, instrumentation, languages, and applications—are compiled and loaded into the Lisp environment. Incremental compilation, supplied by the environment, allows changes in these definitions have immediate effect, even during a simulation run, which is an important capability during debugging.

The application developer or system architect starts a simulation by first instantiating a design corresponding to the particular architectural model under study. The user then chooses a particular instrument and attaches it to the generated design, so that the instrument panels appear on the workstation screen. Another call then 'loads' the application program into the simulated multiprocessor and gets it running, at which time the instrument panels begin to dynamically display the chosen system performance measures. The user is free to interrupt the run both via the keyboard or by breakpoints inserted into the application or model codes. Menu-driven interactions allow variation of component model parameters as well as control of the instrumentation.

1.2 Building System Models

A system model or design is defined in SIMPLE by specifying its intended structure and behavior. As described earlier, this specification is organized around the components that form the system. In this section, we discuss the means by which system models are formulated in terms of components.

1.2.1 Structure

A system structure consists of a hierarchically-organized collection of typed components, as shown in figure 1.2. Defining such a system structure in SIMPLE involves defining each type of component and describing its contribution to overall system structure in terms of its subcomponents and their relationships.

Defining Component Types

A component's type determines its private structure, that is, the set of attributes that make up the component. SIMPLE provides the **defcomponent** macro to define a class of component.[3] This includes specification of its inheritance, and of the named slots, or *instance variables*, present in an instance of the class. Although slots may be used for any purpose, they primarily represent the *state variables* that are required for generating history-sensitive component behavior.

[3]This macro has a straightforward translation to the underlying **defflavor** construct which defines a *flavor*. It also generates definitions for an :init method, invoked by the Flavors system to initialize the slots of a new instance, and for a :reset method, called by SIMPLE when a system simulation is re-initialized.

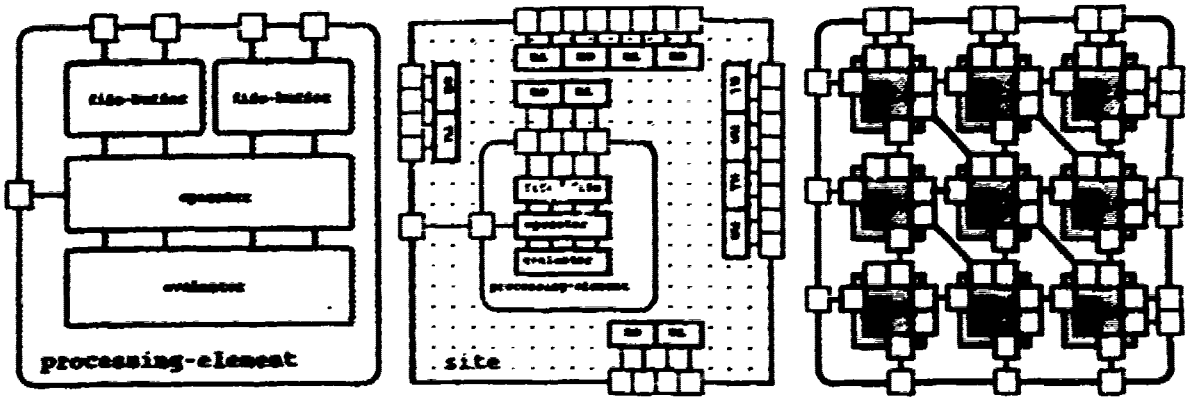


Figure 1.2: Hierarchical Composition

The code in figure 1.3 shows a fragment of the definition of a CARE operator component type.

```
(defcomponent OPERATOR (debug-history-mixin)
  ((Status ; state variable
     :documentation "Current status: ready/busy/servicing"
     :initform 'ready :resetform 'ready)
   (Pending-Operations ; state variable
     :documentation "Data movement operations requested"
     :initform  (make-queue) ; fifo
     :resetform (reset-queue Pending-Operations))
   ...)
  ...)
```

Figure 1.3: Definition of a Component Type

In this example, an operator is made to inherit from debug-history-mixin, thus gaining functionality for keeping a history of events during behavior debugging. Status defines a state variable that contains a symbol that will reflect the run time state of the component. The Pending-Operations slot contains a complex data structure (a queue) that will be managed by the component during its operation.

Defining System Structure

System structure in SIMPLE is built up through the incremental combination of components. Components can form a larger structure through *hierarchical composition* and *interconnection*.

As shown in figure 1.2, at the base level are *primitive* components that have no structure beyond their *ports*. An **operator** is an example of such a component. *Composite* components, such as a **site**, additionally contain subcomponents as *parts*; parts may of course be primitive or composite. Additionally, composite components may have function beyond what can be inferred strictly from their composition.

Composite components also determine the *connections* between the ports of their individual subcomponents, and, further, the connections between their own ports and those of their subcomponents. Connections thus establish pathways for information to propagate between ports: both within and across hierarchical boundaries. Thus, the top-level composite, or *design*, forms the system structure under study.

SIMPLE originally captured component structure graphically and interactively, through the menu actions and mouse gestures supplied by a structural editor. Defined component subsystems would then be placed in a 'library' for later reuse. It turned out, however, that this approach was sometimes inconvenient. Furthermore, a database distinct from the underlying Lisp type database had to be maintained. Therefore, SIMPLE now represents structural information *procedurally*. A procedural representation permits efficient, flexible, and parameterized structure generation. It is particularly useful for automating the construction of the largely replicated system structures that characterize multiprocessor architectures.

A component's structure is specified as a method (that is, a procedure relevant to the type of the component) that is executed by SIMPLE's component instantiation protocol. The method uses SIMPLE functions that create ports, subcomponents and connections to generate a component's structure. SIMPLE also provides additional functions that allow the description of the structural geometry of the component. In effect, then, these functions form the primitives that allow the construction (and querying) of a database of component objects. The protocol accomplishes the creation of a system structure in a depth-first fashion. Components construct subcomponents, which in turn construct *their* subcomponents, and so on until primitive leaf components are created.

To illustrate this approach, consider the code in figure 1.4 that might define the structure of a **processing-element** in figure 1.2.[4]

Subcomponents. Subcomponents are created via the **part** construct, which takes as arguments a name for the part, its type, and, optionally, parameters to customize the creation of the component. Thus, in figure 1.4, **ibuf** will hold a **fifo-buffer** component named **buffer-in** and with a depth of ten items. Once created, subcomponents can be accessed by name through the **part?** function, here, however, they were stored into local variables for convenience. They may also be stored into predefined component slots, or into data structures accessible via slots.

Although the structure shown here does not need it, arguments to be delivered to a subcomponent's structure generation method can also be supplied to **part**. For example, parameters specifying dimensionality and connectivity might be among those passed to a subcomponent that generated a network of nodes organized into a grid topology.

[4] In reality, higher level functions or macros might be used to hide the details of the primitives provided by SIMPLE. For example, the construction and placement of a port might be merged into a single construct.

```
(defmethod (PROCESSING-ELEMENT :instantiate-structure)
          (&key &aux ev op ibuf obuf)
  ;; Construct subcomponents and store into local variables
  (setf ev   (part 'evaluator  'evaluator)
        op   (part 'operator   'operator)
        ibuf (part 'buffer-in  'fifo-buffer '(:depth 10))
        obuf (part 'buffer-out 'fifo-buffer '(:depth 5)))
  ;; Construct ports
  (in  'packet-in)  (out 'status-out) ; for ibuf
  ...
  ;; Establish connections between ports
  (conn (port? 'packet-in)       (port? 'packet-in  ibuf))
  (conn (port? 'status-out)      (port? 'status-out ibuf))
  (conn (port? 'packet-out ibuf) (port? 'packet-in  op))
  (conn (port? 'status-in  ibuf) (port? 'status-out op))
  ...)
```

Figure 1.4: Defining Component Structure

Ports. Ports are classified by SIMPLE as either for input or output. Their corresponding constructors are in and out, both of which accept a name for the port as a parameter. Thus, in figure 1.4, `(in 'packet-in)` creates and returns an input port named `packet-in` for a `processor`. Ports can always be retrieved by name through the `port?` function. An optional argument identifies the port's component; the calling component is the default.

Connections. Unidirectional connections are established through the conn function, which takes two ports as arguments. Connections between subcomponents must be between disparate types of ports: from input ports to output ports. Conversely, connected ports on a subcomponent and its superior must be of the same type, so that information may flow up and down the hierarchy. The information on a connection will be handled by the lowest component in the hierarchy that has an input port accessible via the connection.

Geometry. SIMPLE allows the structure generation code to be embellished with optional descriptions of the geometry of the component structure. Components can then be inspected via a graphical previewer, which facilitates debugging. To accomplish this, SIMPLE provides constructs to define the rectangles representing components, to place ports around the perimeter of the rectangle, to route connections in Manhattan space, and to place, group, align, and geometrically transform subcomponents.

1.2.2 Behavior

A component is essentially a state machine with a notion of time. Its behavior defines the causal and temporal progression of its states and relates this with the rest of the system via its ports. System behavior is therefore no more than the composition of the behaviors of its components.

In SIMPLE, *events* signify the temporal state *changes* in the simulated system, in terms of the changes in the values of the ports and state variables of the system's components. Events make the simulation of large, complex systems tractable, by exploiting the property that only a small fraction of the state variables in the system actually change at any instant in time. This makes it more efficient to keep track of these changes and to compute their consequences, than to recompute the state of the entire system at every time step. An *event-driven simulator* maintains the temporal relationships between events so that time always moves forward.

Within this framework, the behavioral specification of a component is formulated in terms of its responses to the events relevant to it. These responses may include state changes caused in the simulated future, that is, consequent events to be handled by the simulator, as well as direct operations on component state. The assertion of consequent events and the responses to them (involving further consequences) drives the simulation. When there are no more events to handle, the simulation is complete.

To maintain modularity in a simulated system, a component's responses to events should generally be local to it. Consequent events involving a component's output ports are translated by the simulator into events involving the connected input ports of other components. Hence, the effects of a local change propagate between components along the connection paths defined by the system structure. Sometimes, however, a direct, non-local operation on a related component (for example, a subcomponent) might be appropriate. SIMPLE does not prohibit the modeller from accomplishing this.

SIMPLE captures behavior definitions procedurally, as a method on a component class. This method is charged with *asserting* and *processing* the events that drive the simulation.

Asserting Events

In concrete terms, an `event` in SIMPLE is a record that represents a single state change to the simulated system. It stipulates the component affected, its port or state variable changed, the new value it will get, and the (future) simulated time at which it will attain that value. Assserting an event therefore involves generating such an `event` record and passing it to the simulator for later processing

Ports are first-class citizens in SIMPLE, and events are asserted on them by means of the `assert-port` primitive. An output port can be retrieved via the `port?` function described earlier, and can thereby be passed as an argument to `assert-port`, along with its new value and the simulated time of the change.

State variables, on the other hand, are simply *places* (in the `setf` sense [3]) that may hold values. A state variable is therefore specified by the expression that will access the place that holds its value. Thus, for example, a slot denoting a top-level state variable of a component is simply specified by naming the slot. The `Status` slot of an `operator` is such a state variable. As a more general example, if `Registers` is a slot denoting a vector of simulated registers, then the expression `(aref Registers 5)` is an accessor for the state variable representing the register identified. The `assert-state` primitive is used to generate an event on a local state variable. As with ports, there are no restrictions enforced by SIMPLE on the values held by state variables.

Processing Events

An event is passed to the simulator as it is asserted. At the appropriate simulated time, the simulator processes the event: that is, makes the state change specified by the event and then invokes the method that defines affected component's response to the event. The parameters to the behavior method are: a specifier for the port or state variable affected, its new value, and the simulated time of the change (which may be thought of as the 'current' time).

The behavior method defining a component's response to events is typically structured as a set of 'rules'. A rule tests for conditions and, as satisfied, asserts or directly effects consequent actions. The conditions may include arbitrary predicates on the event parameters as well as the state variables of the component.

SIMPLE supplies a number of primitive predicates for testing events. The simplest predicates test if the event occurred on a specified port or state. Others additionally test if the asserted value satisfies conditions such as equality with constants, membership in a set of values, or membership as defined by type. Condition predicates may be combined through Lisp operators such as **and** and **or**.

Modelling Synchronous Designs

Event based simulators are based on the assumption that state and port variables remain unchanged until explicitly modified. Synchronous designs, that is, those in which the opportunities for state change are temporally quantized to a clock, can be modelled in such implicitly asynchronous simulators by asserting the clock signal on a port of each and every clocked component of the simulated system. However, if only *some* of the components in a system need take action on each clock signal (as is typical), there is an obvious inefficiency in this approach that is crippling for systems with even a modest number of components.

If, on the other hand, event times are restricted to integers, the clock can be assumed. All that is needed is a way to detect the event for which a boolean combination of conditions as strobed by an assumed clock is *first* met. SIMPLE supplies primitive condition predicates for detecting an 'edge' (a value changed by the current event) with a coincident 'level' (a value set before the current event) of two ports or state variables of a component in either of the two possible event sequences. The predicate `port-state?` in the example behavior rule shown in figure 1.5 has these semantics.

This code also illustrates the generality of SIMPLE behavioral descriptions. Actions may directly manipulate state variables (as is done to set `Status` to `'servicing`), assert events (as is done to the `Status` state variable and the `Evaluator-Packet-Out` port), call arbitrary procedures (for example, `queue-take` and `time-update`), or call methods (such as `:operation-cycle`). In fact, the last approach has proven to be a natural way to realize the functional operations of components not described by behavioral rules.

1.3 CARE Architectural Models

CARE defines a small number of multiprocessor components, both primitive and composite, along with the data structures manipulated by them in support of the LAMINA concurrent language extensions. These are

```
((and (port-state? Evaluator-Status-In 'free Status 'busy)
      (not (queue-empty Pending-Operations))
      (eq 'to-evaluator
          (operation-place (queue-top Pending-Operations))))
 ;; If the operator is 'busy & there's something in the
 ;; queue for the evaluator & the channel to the evaluator
 ;; is 'free, then pop the queue and transmit.
 (let* ((top (queue-take Pending-Operations)) ; pop queue
        (post-time (send self :operation-cycle top now)) ; when
        (packet (operation-packet top))) ; what
   (time-update packet post-time) ; tir stamp
   (setf Status 'servicing) ; block rur
   (assert-port Evalu--e-Packet-Out packet post-time) ; xmit
   (assert-state Status 'busy (1+ post-ti-e))))
```

Figure 1.5: A Behavior Rule

briefly described below.

1.3.1 Primitive Information Structures

The basic information structures manipulated by CARE components are the *process*, the *stream*, and the *packet*. Processes encapsulate a single thread of application code, and, perhaps, an address space. They communicate and synchronize by operating on streams, which are essentially queues that can store sequences of arbitrary values. Although streams are localized to a single processing site, they may be referenced by remote processes. Typical operations on streams involve treating them as message buffers, that is, sending and receiving messages on them, or treating them as memory cells, that is, reading and writing them Operations on streams and processes are effected by packets of information being communicated between, and interpreted by, the components of the simulated multiprocessor system.

The LAMINA primitives can be used to model shared variable or message passing styles of computation (and variations in between) as described in chapter 2. An application program consists of sequential Lisp code interspersed with LAMINA language constructs that have been built from these primitives. During execution, the primitives cause control being passed to the simulator for their handling. In this way, a simulation achieves its goal of focussing on the interactions between processes.

1.3.2 Components

The component types supplied by CARE are essentially those shown in figure 1.2. They are elaborated upon below.

Communications Components

CARE supplies a small number of primitive communications components that accept (or block), route, and buffer transmissions in accordance with a dynamic, flow-controlled, multicast, cut-through communications protocol as described in the appendix 'A Dynamic Cut-through Communications Protocol with Multicast'. Currently, these are the **net-input** and the **net-output**. Transmissions are encapsulated as packets containing routing information along with control information and application data. To maintain integrity in the simulation, data values transmitted in packets are copied before being passed to the communication subsystem, and packets are sized accordingly.

In keeping with the objective of focusing simulation cycles on the aspects of the simulation particularly relevant to multiprocessor operation, the behaviors of the communications components are defined in fair detail, that is, at the register transfer level. Routing operations are described procedurally and assumed to occur within a time set by a parameter to the simulation. Other parameters allow choice of the routing algorithm used, the width of data channels, and so on.

Processing and Scheduling Components

CARE supplies the **processing-element** to accomplish the processing work of a CARE system. This composite consists of an **evaluator**, an **operator** and a pair of **fifo-buffers**. The storage associated with this component is not explicitly modelled.

The buffers interface the processing subsystem with the communications subsystem and are used for local packet receptions and transmissions. Their behavior is also described at the register transfer level, and allows parametric control of buffer depth.

The evaluator does the real work of the application: executing application code, that is, running processes. The operator does the overhead work associated with such evaluations, that is, managing streams and processes. For example, the operator schedules processes for execution by the evaluator, receives and interprets request packets for operating on local streams (such as queueing messages on them), and constructs packets that require operations to be performed on remote streams and delivers them to the communications subsystem. Depending upon the computation model, then, the operator can function as a message co-processor or as a memory controller.

As indicated previously, the simulation of the operator and evaluator is broken into two aspects: the control of the flow of information and the functions performed on that information. The former is described in terms of SIMPLE behavior rules, register transfer by register transfer. The latter is described directly in terms of procedures, and the simulated time taken by such procedures is modelled.

In the case of the operator, this is done as a function of the number of storage cells manipulated during the operator method that handles some primitive operation. In the case of the evaluator, this is done as a function of the execution time used by the machine executing the simulation, that is, the simulation vehicle. Care is taken to ensure that such overheads as page faults and process switches are discounted in measuring application execution time on the simulation vehicle.

The parameters associated with operators and evaluators include performance parameters such as cycle

times, interrupt times, process switch times, packet formatting times, and so on.

System Level Components

CARE systems consist of a number of `sites` interconnected in some regular topology. Sites may currently be embedded into *mesh*, *torus* and (hierarchically) *bussed* topologies. The basic site composite is parameterized to generate communications components for up to eight 'neighboring' sites; it also contains a local processing element. Specializations of the site, for example, the `torus-site` and the `bus-site`, exist to fit the site into alternative topologies by supplementing the site routing procedures as appropriate to the topology.

1.3.3 Concurrent Application Development

CARE has evolved to provide a number of features that aid in developing concurrent applications. These include:

- Full integration with the underlying Lisp program development tools such as inspectors, debuggers and editors. Components and the data structures they manipulate have abstraction interfaces that provide a summary of their state information when they are displayed in text form. These text abstractions are 'mouse sensitive' and so can be inspected at successively finer levels of detail if desired. Application and model code can be debugged via graphical inspection and manipulation of stack frames. Within the debugger, a single keystroke brings the relevant source code into the editor. Incremental recompilation allows changes to source code to take immediate effect, even within the interrupted stack frame. Thereupon, execution can be backed up and retried, given that intermediate side effecting code is safely re-executable.

- A means for running batch simulations via script files. The script files might contain commands that vary application-specific parameters and data sets, as well as system configurations and parameters—perhaps based on the results of runs previously completed. This facility has been used for experiments spanning several days to weeks.

- A facility for *recording* simulation executions for later *replay* [2]. The only inherently non-deterministic quantities in a simulation run are those that capture the timing of sequential application code fragments on the underlying simulation vehicle. These timings are recorded into a file and may later be used to derive the deterministic behavior of the rest of the system, that is, replay the original run. This can be useful both for debugging and for varying instrumentation for the identical run.

1.4 Building Instrumentation

The results of a simulation are primarily the insights it provides into the operation of the simulated system. The 'insight' we frequently experienced using an early version of the simulation system was that more interesting results could have been produced by the run just completed if only the instrumentation had been

different. With this in mind, the design for the current version of the instrumentation system was aimed at flexibility, while retaining efficiency to the greatest degree feasible.

1.4.1 Abstractions and Implementations

SIMPLE's instrumentation system is organized around *probe*, *panel*, and *instrument* abstractions. Probes *monitor* individual components, and, when appropriate, supply *abstracted* data they have collected to panels Panels *transform* and *save* interesting data from particular kinds of probes in the system, *organize* the transformed quantities along various dimensions, and periodically *display* the results of summary *analyses* on this information. Instruments package together a collection of particular panels, thus providing simultaneous access to different views of operation of the instrumented system.

SIMPLE implements these abstractions by providing a library of classes, methods and procedures that obey a predefined *measurement protocol*. Probes, panels and instruments are built through instantiation of classes derived from the base classes, and the protocol provides the foundation for customizations that allow them to achieve the desired functionality. This is shown in figure 1.6.

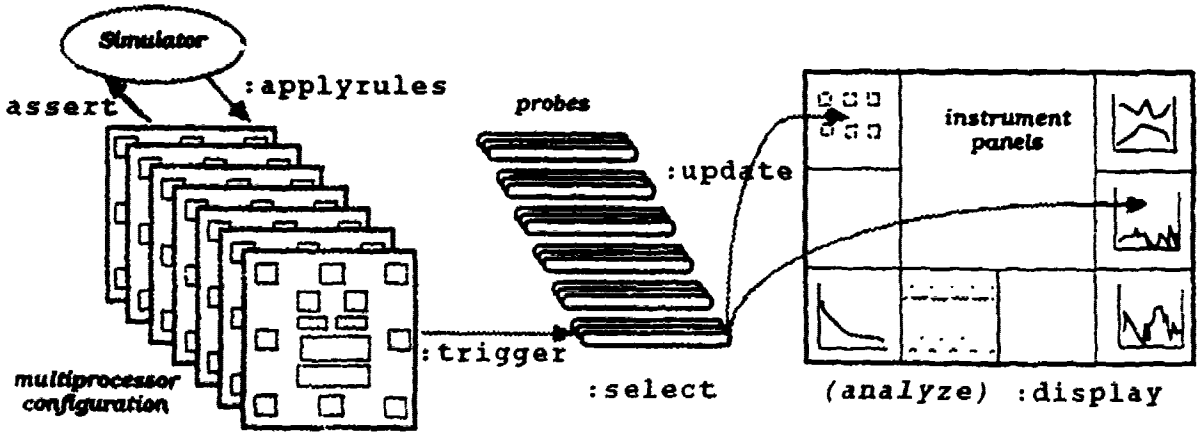


Figure 1.6: Instrumentation Runtime

SIMPLE is designed to make the specification of these customizations as incremental as possible so that existing solutions can be reused. The metaphor it provides to do this is a familiar one: specialization, which is implemented through the extensive inheritance facilities of the underlying Flavors system.

Specializations may range from defining the body of a method invoked by the measurement protocol to providing default values for predefined slots that affect the behavior of the methods that implement the underlying protocol. Default slot values may, in turn, range from simple values such as strings denoting panel legends and functions defining probe filters, to lists representing code expressions that are parsed, compiled and called at runtime by panels to accomplish their transformation, analysis and display operations. While a full discussion of the system-supplied opportunities for customization through specialization is beyond the scope of this paper, the following sections will attempt to show that the design tries to ensure that simple things are simply specified.

1.4.2 Data Capture: Probes

Each probe is attached to a single component in the simulated design and is responsible for monitoring a particular aspect of its behavior. This monitoring is made non-intrusive by ensuring that a probe is informed of all events pertaining to its attached component, as shown by figure 1.6. The `:applyrules` method that defines a component's behavior also has a *daemon* method that invokes the `:trigger` method of all attached probes, passing the event parameters to each. A probe is then responsible for taking action based on the event, if desired.

Probe actions may involve filtering events, querying the values of ports and state variables on the attached component, manipulating the contents of the probe's own instance variables (so that probes can be history-sensitive), and, finally, processing and forwarding data to attached panels. Processed data is formatted as a property list that tags each data value with an identifying keyword symbol, and it is encapsulated with a *probe key* signifying the semantics of the data, a *probe object* for which the data is relevant, and a number representing the simulated time. The probe object can be an arbitrary data structure such as: the attached component or one related to it (for example, the component enclosing it), a data structure manipulated by the component (for example, a process structure of an evaluator), or even an 'application level' data structure (such as a LAMINA object).

An Example

As an example probe, consider the `evaluator-queue-probe` defined in figure 1.7. This probe measures the load on an evaluator in terms of the number of runnable (and running) processes queued on it. Since processes arriving from the local operator (via the `Packet-In` port) increment the load, and since transitions in the evaluator's `Status` reflect the status of the process currently being run and thereby affect its load, the `:trigger` method checks to see if the event on the attached evaluator is relevant before taking action. This is done through the `state-event?` and `port-event?` predicates.

The declarations of this probe's instance variables use the `probe-state` primitive to cache static slots in the evaluator. In general, instance variables are used to store intermediate state as required for probes that track interesting *sequences* of state changes (for example, the scheduling transitions of a process). Note that supplied data is tagged with the `:evaluator-queue` probe key, and that the site component that contains the evaluator is passed as the probe object for which the data is relevant. Note also the conversion of event time units (`now`) into model-specific time units, through scaling, as the data is passed on to panels.

```
(defprobe EVALUATOR-QUEUE-PROBE  () ; no mixins
  ;; slots that cache the attached evaluator's slots
  ((Input-Queue (probe-state Evaluator-Queue))
   (Site        (probe-state Site)))
  ;; options
  (:documentation "Report evaluator process queue lengths")
  (:component-type evaluator) ; attach to evaluators
  (:probe-key :evaluator-queue) ; data tag
  (:trigger (tag value now)   ; name the event parameters
    (when (or (state-event? Status)     ; status change?...
              (port-event?  Packet-In)) ; or process arrival?
      (send self :select    ; i.e. inform attached panels...
        Site                ; probe object = site component
        :evaluator-queue    ; probe key
        (list :busy         ; probe data property list
              (+ (queue-length Input-Queue) ; # enabled processes
                 (case (probe-state Status) ; running process?
                   ((ready stalled) 0) (t 1))))
        (simulated-microsecond-time now)))))  ; probe time
```

Figure 1.7: Example Probe Definition

The `:select` method is part of the measurement protocol for probes. It forwards the probe data on to selected attached panels by calling the `:update` method of the panels. Selection is done via a filter that can be specified in **defprobe**; the default is to pass data through to all connected panels.

1.4.3 Data Analysis and Presentation: Panels

Panel operations are accomplished by successive transformations on the data supplied by probes, ultimately yielding the quantities that are displayed along the various 'axes' defined by the presentation style of the panel. These transformations are conceptually accomplished through manipulations on two kinds of records:

- a *state record* for each probe object, that stores relevant information derived from the probe data passed in.
- a *display record* that stores the quantities that need to be displayed, and forms the foundation for display lists.

Display records are organized along panel-specific dimensions to satisfy display goals. These may be times, durations, frequency and counting bins, probe objects, and so forth. Both state and display records are created as needed by the panel.

The actions taken by a panel are then to:

- ***update*** the state and display records corresponding to the probe data passed as parameters to the **:update** call. This involves extracting the required data from the probe data property list, computing transformed values based on this and the retained data stored in the records, and storing results back into the appropriate records.
- ***analyze*** the display lists periodically, that is, reorganize them based on display objectives, such as sorting on display record fields.
- ***display*** the results of these periodic analyses in the display style of the panel, that is, transform display list quantities into graphics actions on the screen.

At the base level, SIMPLE provides *presentations*: class definitions that represent particular display styles. SIMPLE's current class library includes scrollable text displays, scatter plots, fixed and scrollable line plots, histograms, strip charts, intensity maps and signal animations. These are customized through specialization to define panels.

Customizations include those that affect a panel's graphical appearance, such as legends, scales, axes labels and the like, as well as those that achieve its functional objectives. The latter include declarations of the types of probes required to drive the panel, and *interface specifications*: arbitrarily complex expressions that specify the transformations between the information provided by the probes and that saved and displayed by the panel. Other customizations control the computing resources used by the panel; these are parameters such as sampling intervals, refresh periods, and history depths. Presentations have been defined so that they supply the most commonly required customizations implicitly.

To retain run time efficiency, the transformation expressions that are stipulated in the panel declaration are processed when a new instrument is created. They are compiled at that time into code bodies referenced by run time control blocks associated with the underlying methods that implement the panel measurement protocol.

An Example

As an illustrative panel definition, consider the code shown in figure 1.8. This defines a strip chart that plots the history of total evaluator queue lengths in the system over time, thus providing a view of the available application concurrency.

The important points about the specification are:

- The **probes** slot, which specifies what probes are required and how the data supplied by different probes will be mapped into the transformation expressions that use the data. This generalized binding format allows the panel to distinguish or combine data from different types of probes, as needed. Keeping probes isolated from the transformation expressions in this way allows different probes to be 'plugged in' to the panel by simply specifying a different binding list. The resolution to actual probe keys will be automatically done when the panel is instantiated and its code expressions processed.
- The **-axis-form** slots, which contain expressions describing the transformations on probe data. Within an expression, the general form for denoting keyed data values supplied by a probe is as a list composed

```
(defpanel EVALUATOR-QUEUE-HISTORY-PANEL
  ((name    "EVALUATOR QUEUE HISTORY")
   (legend  "Total Evaluator Queue Lengths")
   (time-scale-factor  0.001)  ; us [from probes] to ms [display]
   (sampling-interval  200)    ; 1 sample kept per 200us
   (scroll-range       10)     ; 10ms 'window' of time displayed
   ;; interface specifications
   (probes           '((:queue-probe evaluator-queue-probe)))
   (left-axis-form   '(:queue-probe :busy save-sum)) ; queue lengths
   (bottom-axis-form '(:simulator :time)) ; reported probe times
   (plot-update-form '(send self :update-time (:simulator :time))))
  ;; axes specifiers
  ((left-axis   (make-axis :label "Evaluator Queue Sum"
                           :range (make-range 0.0 nil))) ; open ended
   (bottom-axis (make-axis :label (format nil "MS by~DUS"
                                          sampling-interval)
                           :range scroll-range))) ; fixed range
  (scrolling-line-plot-presentation)) ; base SIMPLE presentation class
```

Figure 1.8: Example Panel Definition

of an abstract probe key and the relevant data key, such as `(:queue-probe :busy)`. These value expressions can be combined with others as required (through arithmetic or user-defined functions) to compute derived values. For example, one definition of 'load' on a resource in CARE is through the formula $1-(1/1+Q)$, where Q represents the sum of the lengths of all the queues providing the resource with work, which might be expressed as

```
(- 1.0 (/ 1.0 (1+ (:queue-probe :busy))))
```

The optional `save-sum` modifier in the probe value expression found in the `left-axis-form` introduces a summation transformation, which requires that the overall sum be decremented by the previous `:busy` value reported for the probe object, and then be incremented by the new reported value. Were the modifier absent, the relevant display record would simply reflect the latest value reported; instead, it now maintains the running total of the latest reported values per probe object. SIMPLE has a number of such *save functions* to aggregate and classify data for display; it also provides a means for new ones to be defined.

- The `:update-form` slot, which ensures that the panel organizes display lists along the dimension of simulated time, corresponding to the 'bottom axis' of the display. In general, this needs to be specified only when mapping time; otherwise, the default update behavior is sufficient.

This panel does not need the analysis feature that most panels provide as an option. SIMPLE's basic analysis operation allows sorting display lists by arbitrary predicates applied to arbitrary record fields. This is expressed through an `analysis-form` declaration such as

```
'(sort-arrays
   (list (list #'> (:latency-probe (+ :launch :network)))))
```

This code specifies that display records are sorted in decreasing order of the sum of the 'launch' and 'network' delays reported by a 'latency' probe (presumably monitoring communication latencies). The list of lists format of the specification allows for progressively finer sorts on identical items.

1.5 Understanding Instrumentation in CARE

In this section, we will try to show how instrumentation helps understand the operation of concurrent CARE systems. To do this, we will focus on a particular programming model—the LAMINA object-oriented model, and its corresponding multiprocessor model—a message-passing CARE multicomputer.

1.5.1 Monitoring Computations

In the LAMINA object-oriented model, an application consists of objects that interact only by asynchronously passing messages containing data values. Objects execute the messages arriving on their local task streams serially. Each message execution, or task, atomically manipulates the message contents and the object state and then sends new messages, thus continuing the computation at some other object.

The CARE message-passing machine model provides the resources that accomplish the LAMINA computations described above. Evaluators run the processes that execute the LAMINA object tasks. When a task needs to send a message, the evaluator interrupts the local operator and passes it the message data. The operator encodes the data values into a packet and passes it to the communications components for remote delivery. These route and deliver the packet to the remote site according to some communications protocol. The operator at the target site queues the message packet on the relevant task stream and perhaps reschedules a waiting object process for execution in the local evaluator.

A LAMINA application can thus be effectively monitored by simply monitoring the critical operations performed on messages by LAMINA objects: the *generation* of messages, the *arrival* of messages on the target object's task stream, and the *execution* of messages. Its performance can then be understood by monitoring the actions performed by the underlying system resources in supporting this message traffic: the *creation*, *communication* and *receipt* of packets, and the *scheduling* and *execution* of processes.

This captured information provides a basis for understanding system operation. The impact of the application decomposition can be studied in terms of task and message granularities, message volumes frequencies, over- and under-utilized objects and classes, and so on. The impact of the system design, its operating parameters, and its finite resources can be studied in terms of resource utilization, service latencies, resource conflicts, load imbalances, resource bottlenecks and so on. Some examples of these are given in the next section.

1.5.2 Seeing System Activity

In this section, we will describe a few representative panels that illustrate the visualization capabilities of CARE's instrumentation

Activity and Load Maps

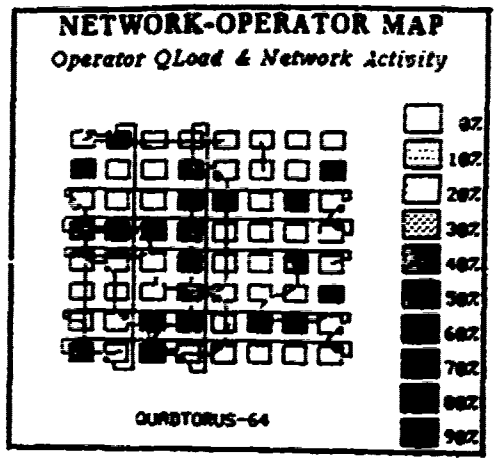


Figure 1.9: Mapping Panel

One of the most intuitively useful kinds of presentations is the *mapping panel*. It provides an animation of activity in the design—seen in terms of the spatial arrangement the system designer laid out when the structural organization of the design was defined. In the "network operator map" panel shown in figure 1.9 uses this topology to display operator loads and network activity.

The boxes in figure 1.9 correspond to operators in the system. Their shading indicates how many packets are queued up for service by the corresponding operator. As the simulation proceeds, the indicated load on operators shifts, so that bottlenecks in operator resources stand out visually. Load imbalance on operators show up as more or less constant utilization of only certain operators.

The lines between boxes correspond to connections made for packet transmission between the network ports of the indicated sites, so that a qualitative view of the degree to which the network is utilized at a given time in the simulation is available. We have found this useful in debugging the network protocols that we have experimented with—deadlocks and thrashing are immediately apparent.

Mapping presentations have been specialized to depict a number of different measures. We have found it useful for seeing object load (the number of LAMINA objects at a site), message load (the sum of the lengths of all LAMINA task queues at the site), evaluator load (the number of runnable processes at a site), or, simply, evaluator status.

Utilization Histograms

A more statistical view of the operation of the system over time is provided by the presentation of, for example, the utilization of operator and evaluator resources as *histograms*. The "processor utilization" panel shown in figure 1.10 has been specialized to show the percent of the time during a simulation a given number of evaluators (displayed on the top half of the panel) and a given number of operators (displayed on the bottom half of the panel) have been used. Highlighting shows what the current situation is as well as what the average situation has been through the current time of the simulation

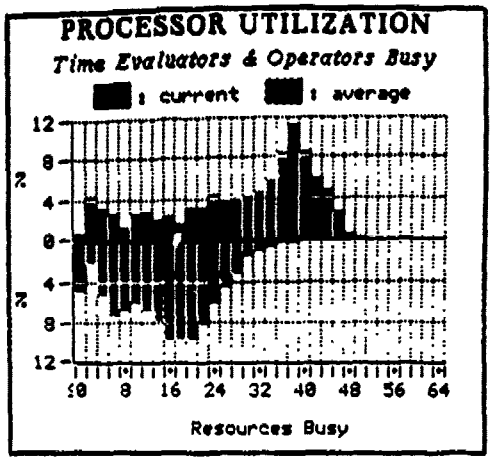


Figure 1.10: Utilization Histogram

If it turns out that only a few evaluators are concurrently active, it may be that the application is not generating enough concurrency, or that the load is unevenly balanced, so that available concurrency is not exploited. Other panels, described below, may be used to clarify this explanation.

Load and Latency Strip Charts

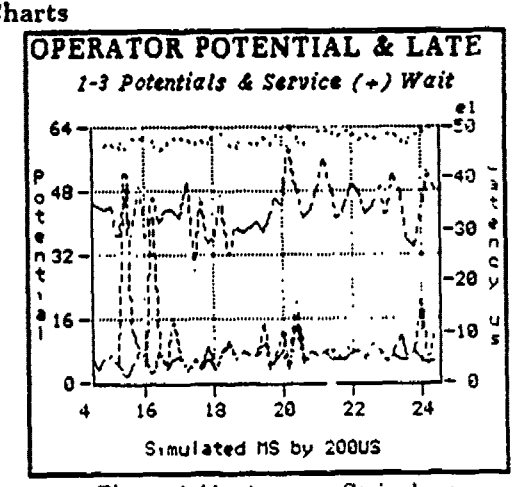


Figure 1.11: Activity Stripchart

A *strip chart* presentation is often a useful way to see what the history of some measure of system activity has been in the recent past. There are four such measures plotted in 'operator potential & latency' panel shown in figure 1.11. Two of the measures are plotted on the scale at the right side of the strip chart. These show the latency being experienced by operators in the system as they receive and service packets from the network. The time to service packets appears to be relatively constant in the plot shown. As shown, there are occasional delays between the time a packet is received and service on it is begun. This delay is shown as an offset on top of the time plotted to service packets.

If latencies have a periodic character with unacceptable peak times, it may be that there are load imbalances that can be addressed to improve this situation. Alternatively, more or less monotonically increasing latencies indicate pipelines that are not keeping up with their inputs. If the affected pipelines can be replicated and

work spread among them or the grain size of the larger pipeline stages reduced—and the resources dedicated to all these pipelines and pipeline pieces are adaquate to the demand—the bottleneck causing the increasing latencies may be broken.

The two other measures plotted in figure 1.11 refer to the 'potential' for additional work remaining in the system. Their scale, indicating the number of operators not in use, is shown on the left side of the stripchart The lower of the two plots indicates the number of operators that have no packets in their service queues. The remaining measure plotted is like this one: it indicates the number of operators that have less than three packets in their service queues.

The values of these potential plots is an indication of resource utilization over time. The distance between them is an indication of load balance. if they are well spread, most of the resources so described have one to three packets to handle. This is an indication of good load balance. Alternatively, both plots close together toward the middle of the axis indicates that half of the resources described have more than three packets to handle and half have none: an indication of poor load balance. Both plots drawn down toward the bottom of the panel may indicate an overloaded system: all the resources being measured in this way have several packets in their service queues.

As shown in figure 3.2, similar panels have been defined for the communications subsystem (the 'network load & latency' panel) and the processing subsystem (the 'evaluator potential & latency' panel). These can be used together with the one described here to see the relative granularity among the subsystems and their relative utilization, so as to discover the critical resources in the system

Activity Tables

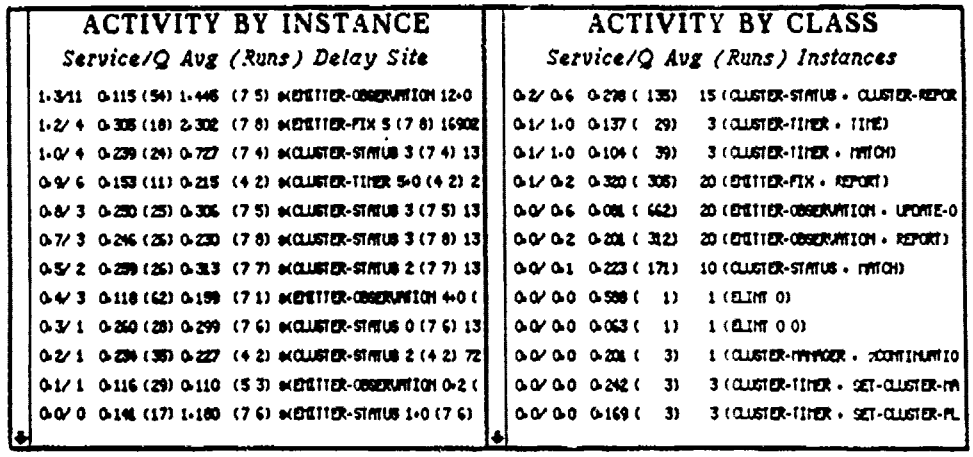


Figure 1.12: Activity Tables

Sometimes, the most informative way to show information is as text. The two panels shown in figure 1.12 use *scrolling text presentation* to dynamically summarize the activity of LAMINA objects in the system in the form of tables.

In the 'activity by instance' panel, each line in the scrolling display represents a single LAMINA object. The columns of the tables denote, respectively:

- the expected service time of the object, that is, the product of its average task execution time and the number of messages in its task stream. This is an indication of the degree to which this object is a bottleneck. The text lines are periodically sorted so that the objects that have the highest service time bubble to the top of the display. Objects forming potential bottlenecks are thereby evident.
- the number of messages on the object's task stream.
- the average task execution time for this object, that is, the average time it has taken to process a message up to this point in the simulation.
- the number of messages that have been processed by the object, an indication of its relative activity.
- the delay experienced by the most recent message that was executed (as a task) by the object. This delay is the interval from the generation of the message by the sending object at some remote site to the actual execution of the task by this object at its site. It represents the overhead involved in getting the task accomplished, and, as such, includes the latency in getting the message delivered as well as the scheduling delay before the process corresponding to the object acquired the evaluator.
- the site at which the object is located. This can be used to discover if the object is bottlenecking because of load imbalance: this is apparent if the most backed-up objects are colocated.
- a printed representation of the object, showing in particular its class. Text lines are 'mouse sensitive': the LAMINA object can be inspected through a simple mouse click

The 'activity by class' panel presents this information averaged by the object class and type of message, as shown in the rightmost column of the display. This information can be used to see the distribution of work in the application design. An inappropriate distribution may indicate the application needs to be reorganized; the display provides information about where this effort should be concentrated.

Cumulative Latencies

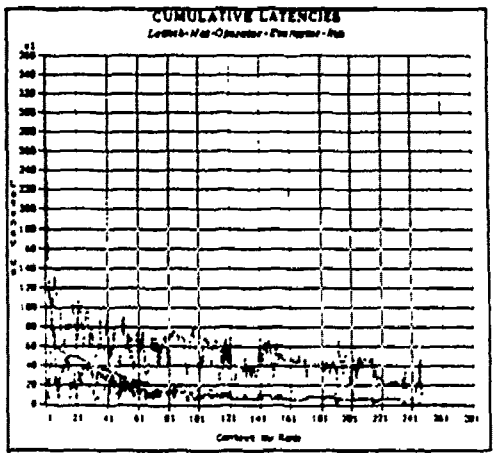


Figure 1 13. Cumulative Latencies

The 'cumulative latencies panel' in figure 1 13 is an example of a *line plot presentation.* It displays a snapshot of the message delays described above, as experienced by the most recent messages received and

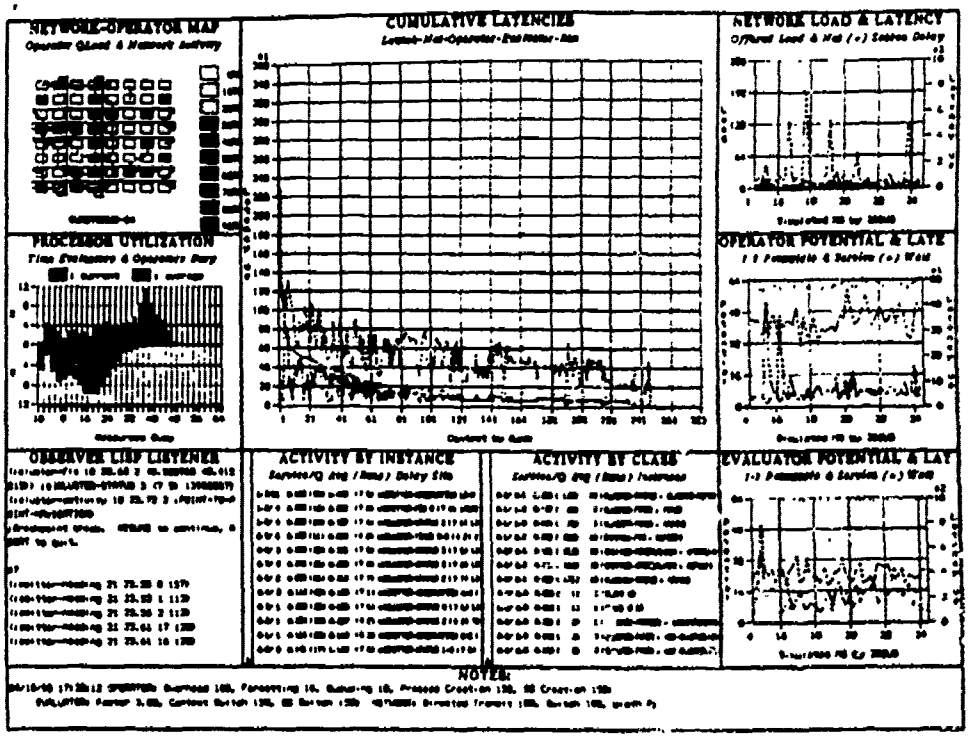


Figure 1.14: CARE Observer Instrument

processed by each of the extant application objects. There are five curves incrementally showing the latency experienced by the messages at the source operator, being routed in the network, waiting for service at the target operator, being serviced by the target operator, waiting for execution at the target evaluator, and, finally, the execution time of the task that consumes the message. The curves are ranked by the sum of the first four delays above, which represents the overhead in getting the requested task accomplished at the targeted object.

1.5.3 A Complete Instrument

The panels described above have been collected into the CARE observer instrument shown in figure 1.14. Additionally, the instrument allows annotations reflecting system parameters and other data, so that experimental parameters are evident.

The instrument thereby provides a unified view of system operation that correlates the activity of hardware abstractions, that is multiprocessor subsystems, with application abstractions, that is, LAMINA objects.

Chapter 2

LAMINA Programming Models

LAMINA is an experimental programming framework that allows concurrent algorithms to be expressed using both value-oriented and reference-oriented styles. It provides mechanisms and syntax (as extensions to Common Lisp [3]) to describe and control concurrent computations so that their performance may be studied using the SIMPLE/CARE architectural simulator. This chapter describes the LAMINA *functional, object oriented message passing* and *shared variable* programming models, along with examples of their use. It also describes the underlying primitive operations that support the models. With this description in mind, you will have the necessary basis for reading the next chapter—which describes how SIMPLE/CARE is used to understand the performance of an already written application program.

2.1 Introduction

2.1.1 Cells, Futures and Streams

Cells form the basis for perhaps the simplest form of interprocess communication and synchronization. Communication is accomplished by reading and writing cells that are shared between concurrent processes. Synchronization can be accomplished by having the memory system support an atomic *read-modify-write* cell operation (such as an exchange).

Futures can be thought of as cells that represent promises for potentially unavailable values. They can be used as placeholders in a computation while their values are being *eagerly* produced by concurrent evaluations for consumption as available. Futures therefore embody both communication (of the produced value) as well as synchronization (because the value must be produced before it can be consumed). *Streams* generalize futures by representing *sequences* of eagerly produced but potentially unavailable values as a single abstract data type. Streams can thus be used to build pipelines of computation connecting the producers and consumers of values.

Streams and futures may be the arguments to or the results of function applications. Furthermore, certain

operators (sometimes called *non-strict* operators) do not require the actual values promised by a stream or future in order to perform their work. For example, a constructor (such as `cons`) may create data structures that include streams as elements without accessing any of the promised values the streams represent, referencing the placeholders is sufficient.

LAMINA provides the `stream` as its primitive data type; a `future` is a specialization of a stream that represents only a single value. Streams and futures, because they represent arbitrary values such as lists and vectors, must be managed by a resource such as a processor—with attendant costs. Cells, however, can hold only single, fixed-size quantities such as small integers or references to other cells; thus, operations on cells (such as `read` and `write`) can be efficiently handled by simple memory controllers.

2.1.2 Multi-Level Address Spaces

LAMINA's address space design is based upon expectations about the expenses involved in global storage reclamation. If references (pointers) are allowed to exist between processor address spaces, relocation of the referenced data (for example, as required by a copying garbage collector) requires global synchronization, which can be expensive. LAMINA's multi-level addressing scheme therefore creates inter-processor references only as necessary, so as to allow for independent, globally unsynchronized storage reclamation to the greatest extent possible.

As shown in figure 2.1, an application's address space consists of

- *static space* containing data structures such as code bodies and constants (e.g., keyword symbols), which are regarded as immutable. They are therefore neither relocatable nor reclaimable, and so may be freely referenced and cached by any processor. LAMINA does not explicitly model transactions concerning data in static space; it assumes that static data is always available in a processor's cache.

- *dynamic space* containing cells (in the shared variable model), and *indirect references* to streams and futures. Indirect references may be thought of as remotely unreadable and unwritable 'reference cells' containing pointers to local data structures that represent streams and futures. References to data structures in dynamic space are allowed to exist between processor address spaces; hence, the data structures may only be relocated through globally synchronized operations affecting all computations that could access them. Note, however, that streams and futures (and the data values that they represent) may be locally and asynchronously relocated because of the indirection involved when they are remotely referenced.

 Streams, futures and cells are *only* visible as references in LAMINA. In the remainder of this discussion, then, the terms 'stream', 'future' and 'cell' should be taken to be equivalent to references (perhaps indirect) to data structures of the appropriate type.

- *local space* containing arbitrary local data values. Local data structures cannot be remotely referenced and are always copied between processor address spaces. They may therefore be independently reclaimed and relocated.

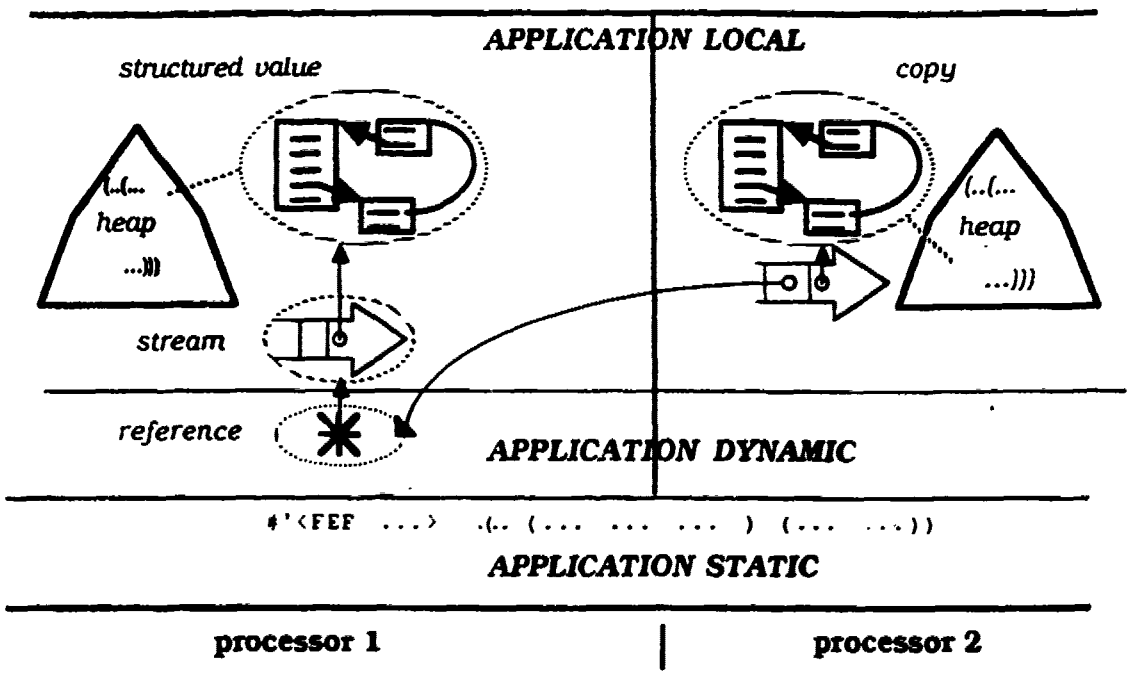


Figure 2.1 Local, Dynamic, and Static Addresses

Communicating Values

In LAMINA, a data structure of arbitrary complexity can be supplied as a value of a stream or future either local or remote to the processor address space in which the structure was generated. This is passed by copying, so that the structure is isomorphically reproduced at the target stream or future.

When values are passed between processor address spaces, the structure representing the value, that is, the *structure value*, is recursively encoded until a data structure is produced which has the same form and internal relationships as the original value but which holds only: *static references* to structures in static space, *dynamic references* to structures in dynamic space, *internal references* to elements of the new structure value, and *self-referentials* or 'immediate' data objects such as small numbers. This encoded data structure thus contains all the information required to form a copy of the original structure at the target stream or future, through the reverse operation of decoding.

Depending on the underlying system, encoding of a structure value might be done asynchronously with evaluation of the user application, so if changes are to be made (at any depth) in the structure passed between address spaces, independent copies of the structure should be formed.

An example of values and references passed between processor address spaces is shown in figure 2.1. One of the values of the stream in the application's processor 2 local address space is an independent copy of the structure value in the application's processor 1 local address space. Both structure values are heap allocated from independently managed heaps in separate local spaces. The other value shown for the same stream is an indirect reference to the other stream; the stream, in turn, represents (or *contains*) the original structure value.

Storage Management

The cheapest approach to the dynamic allocation (and deallocation) of memory is *stack-based* and local. However, the benefits of stack-based operation come at the cost of a prescribed order of deallocation. Additionally, at least for the commonly used memory management enforced stack limit schemes, stack-based operation entails a minimum storage commitment that is significantly larger than the rest of the execution environment for each small granularity evaluation expected for LAMINA programs. Stack based allocation can be used whenever references to structures with dynamic extent [3] are known to be entirely within a given sequential computation.

The next cheapest approach, for references that are local with indefinite extent [3], is heap based allocation in *local* space. Since such references are confined to a single processor address space, their referents may be allocated, relocated, and reclaimed asynchronously with operations on other processors and memories, based on just the information in the associated processor address space.

Finally, as the most expensive approach, global references may be made to dynamically allocated references (that is, to cells and reference cells) which must be relocated under a global synchronization scheme. Allocation in dynamic space is done independently by each processor and each allocation is distinct. Operations involving dynamically allocated references are handled by the processor (or memory controller) associated with the reference. The referents for such references (that is, the streams and futures) are mutable, and may be viewed as uncacheable.

References to locally allocated structures can also be passed between processor address spaces, by encapsulating them in streams and then passing out the (indirect) reference to the stream. By this indirection, pointers to locally allocated structures are held locally (and may readily be relocated) but a means is provided to reference them in other processor address spaces.

2.2 LAMINA Primitives

2.2.1 Creating Streams

Streams are created by the primitive function `new-stream`, which returns a reference to a new stream on the executing (that is, local) site. Futures—streams that have at most one value—may be created by the function `new-future`. Streams and futures may be labelled for debugging purposes by including a 'tag' as the optional first argument of its constructor, as in

```
(new-stream 'requests)
```

The default is for a stream to inherit a tag identifying the execution which creates it.

As described earlier, streams and futures are only visible as references. The *site* of a reference, that is, the processor on which it was created, may be determined by executing

`(reference-site` *reference*`)`

which returns a *site identifier* that may be used to specify sites as required for parameters to other LAMINA primitives. References can also be tested to determine whether they are the 'equal' by the function `eq-reference`, a predicate that tests if the two supplied references are to the same, potentially remote, stream or future.

A stream may be thought of as an ordered queue of `postings`, each containing, among other things, a value. The default order of postings on a stream is non-deterministic arrival order. Sometimes, however, it is desirable to override this default so as to control the order in which values are consumed from the stream.

A stream ordered by increasing numeric keys, supplied as part of the postings it receives, can be created by the function, `ordered-stream`. This is typically used to prioritize the values currently available on the stream. Similarly, a stream that provides values in strict sequence according to non-decreasing integer keys, again supplied as part of the received postings, can be created via `sequenced-stream`. this is typically used to minimize scheduling overhead by deferring executions involving the consumption of 'out of order' values. Both these kinds of stream have application in the LAMINA object oriented programming model discussed in section 2.4.

When a locally allocated data structure needs to be passed between potentially concurrent computations as a reference rather than as (a copy of) its value, the form (`reference` *item*) returns a reference for the value of the item. This is implemented by placing the value on a local *valued stream* which can then be remotely referenced.

2.2.2 Producing Values for Streams

Streams acquire values as a result of postings received by them. This is directly done by a producer using the **posting** operation as in

(**posting** *value* **to** *targets* ...)

The operation is *non-blocking*; it immediately returns and the actual transmission of the (copied) value will occur some time later.

The posting may be multicast by supplying a list of target streams rather than a single target, so that each will receive a unique copy of the value. Additionally, there is a facility for *specializing* the value transmitted in a multicast to the individual targets of the posting. Any place a stream is used as a target of a **posting**, it may be replaced by a cons of that stream and the value specialization for that stream. The value specialization will be prepended to the supplied value and the combined list will be taken as the value of the posting when it arrives at the target stream. Specialization is specified by a list of lists even if only one target is involved, in order to distinguish it from a list of unspecialized targets.

The keys required for correct operation of ordered and sequenced streams can be included in postings by specifying a number following the keyword 'by' in the call creating the posting. Other keywords are also available, and, since they are used by many of the LAMINA primitives, they are listed here.

- **to**, on *targets*: A target stream or list of targets streams for the indicated primitive LAMINA operation. Some primitives expect site targets rather than stream targets, as discussed in later; for these, if no site is provided and one is needed, an unspecified site is chosen. The choice between the alternative keywords shown is purely stylistic.
- **for** *clients*: A stream or list of streams acting as the continuation of the computation that will be triggered by the LAMINA operation.
- **as** *tag*: Arbitrary data for debugging. Defaults to the tag of the sending execution.
- **by** *order-key*: A number which may be used to order information in target streams.
- **after** *delay*: A positive number indicating the number of miliseconds that the operation will be delayed before being attempted.
- **with** *properties*: Arbitrary data intended for user extensions of the posting protocol.

2.2.3 Consuming Streams

The primitive **first-posting** returns the first posting of those present on the referenced stream. The primitive **next-posting** does the same but also removes the posting from the stream. Finally, **last-posting** returns the last posting and eliminates all others on the stream.

If the stream is empty, the three stream posting access functions return **nil**. Otherwise, they return a posting as a list consisting of the *value*, *clients*, *key*, *tag*, *origin*, and *properties* of the posting. For convenience, these

elements of this list may also be accessed by the **posting-** primitives: **-value**, **-clients**, **-key**, **-tag**, **-origin**, and **-properties**. The number of postings available on a referenced stream is returned by the primitive **postings**.

If it is desired that execution be blocked until there is a posting for a specified stream, the stream posting access forms above may be wrapped in an **accept** construct, as in

(accept (next-posting *stream*))

In this case, when a posting is available on the indicated stream, the posting is returned to the restarted or resumed execution.

Futures in LAMINA are defined so that their value, once attained, cannot be removed. Hence only the **first-posting** operator is a valid accessor for a future.

The access primitives described above will, if necessary, coerce the referenced stream into one local to the calling site (through **relocating** as described later). Sometimes, this is not the desired behavior, so a way is provided to access potentially remote streams without incurring this side effect.

Remote Streams

Posting-by-posting access of the information on streams may also be accomplished by requesting that a stream access function be applied to the streams at the site they exist on, as in

(accessing *access-function* **on** *targets* **for** *clients* ...)

The *access-function* may be any of the stream posting access functions, for example, the function **next-posting** described previously. A posting will be sent to the client streams when one is available on a target stream. This is the only way provided for expressing competitive access to a common stream.

An interlocked operation on streams is provided by

(exchanging *value* **on** *targets* **for** *clients* ...)

This causes **last-posting** to be applied to each target stream and the result sent to each client stream. The *value* replaces the last posting on the target stream. The exchange is atomic with respect to each stream.

2.2.4 Managing Streams

Streams in LAMINA may be managed in various ways across the system

Copying Streams

A posting sent to parent streams in a tree of streams set up by copying operations will result in copies of that posting also appearing on all the descendant streams in the tree. Such a system of streams can be built by

(copying *parents* to *children* for *clients* ...)

The references for the child streams are sent in an operation request posting to the parent streams where they are added to the child references of the parent. The current queue of postings held in the parent stream is copied and returned in one combined posting that is multicast to the child streams. These postings become part of each child stream. When each child receives the combined postings, it sends on to the client streams a completion posting whose value is the parent stream from which it received the posting queue. This can be used to validate that a requested copy operation has been accomplished.

Linking Streams

The linking operation is an optimization of copying for those cases where it is known that postings need not be retained on intermediate streams in a system of linked streams. Linking parent streams to child streams serves to restrict the parents to act only as intermediaries in a system of linked streams as in

(linking *parents* to *children* for *clients* ...)

The references for the child streams are multicast in an operation request posting to the parent streams. When a parent receives the references, any postings already on parent streams are sent to the children specified by the references and eliminated from the parents. Further postings are not retained on parents after they receive a linking directive but are immediately passed on to the child streams. For efficiency in forwarding, the implementation may bypass intermediate levels in a system of linked streams.

Value Specialization

Target specialization may also be used with the linking or copying operator to specialize the value of postings transmitted from parents to children as in

(linking *parents* to (list (cons *child-1 value-specialization-1*) ...) ...)

Thereafter, all postings that traverse the links from parents to children will have the appropriate value specialization prepended to their value. This is the mechanism used to support the implicit continuations provided by the LAMINA object oriented model.

Relocating Streams

A `linking` operation does not change the way that a child stream orders postings or presents them. Relocating a stream from one site to another while preserving its accumulated postings as well as its means of ordering and presenting them, is specified by

(`relocating` *parents* `to` *children* `for` *clients* ...)

This is used when there is an attempt to read from a stream that is not local to a site. The attempt causes the reference used to specify that the target stream target a new child stream, the relocation of the previously specified target. No change can be detected in the operation of `eq-reference` on the reference after relocation.

Group Streams

An application in LAMINA may wish to view a group of streams as a composite, carrying out some operation only when all of the streams in the group have received a posting. To minimize unproductive scheduling, computations may wait on such composite *group streams* rather than on the individual streams. Group streams are created by `new-stream` called with a `:group` keyword argument as in

(`new-stream` *tag* `:group` *member-streams*)

A stream may be the member of only one group but a future, since its value, once attained, cannot be removed, is not so restricted. If streams of values are to be made available to several groups, a system of linked or copied streams must be used to accomplish this.

If a member stream is not local to the site of its group stream, a local member stream is created and the remote member stream is relocated there. The postings sent to the local member streams are taken from the member streams whenever a request that has been made to accept a posting from a group stream can be satisfied. Each posting available from a group stream will contain, as its value, a list of the postings received by its component streams. The order of posting elements in the list representing a group posting corresponds to the order indicated in specifying the component streams of the group stream when it was formed. Group streams are used to schedule an implicit continuation only when values are available on all streams upon which the continuation is waiting.

2.2.5 Creating Processes

Restartable Processes

A separate, concurrent computation is created by spawning the execution of a closure as in

(`spawning #'(lambda ()` *form*) `on` *site* `for` *clients* ...)

The closure is formed and the clients returned immediately as the value of the spawning operation. The closure will sent to the indicated site and eventually executed there. The result of that execution will be returned to the specified clients.

Spawned computations can block waiting for a value to be available on a stream. When the value is available they will be *restarted* and any intermediate computations done previously will be redone. This approach is taken to avoid dedicating stacks for every spawned computation. However, often the continuations of partially completed computations can be spawned on the same site as their parent, thus preserving intermediate work as well as eliminating the need for dedicated stacks. This is described in sections 2.3 and 2.4

Resumable Processes

If an execution is blocked on trying to access an empty stream, it can either be restarted, as discussed earlier, or *suspended* and *resumed* when that stream receives a posting. In general, suspending and resuming a computation (without spawning continuations) requires preserving indeterminate amounts of intermediate (control and binding) state with one or more stacks. Maintaining many independent stacks is certainly an expensive operation in simulation and may also be so in a target system implementation.

However, for occasions when the full power and expense of stack switching is warranted, LAMINA provides the **mounting** primitive. This is called and behaves like **spawning**, except that it creates a process with associated stack storage at the indicated site.

One could implement a multiple fork and join construct (like **cobegin** and **coend**) by mounting a number of processes with a common client stream. The creator could then wait for the appropriate number of responses on the client stream (to ensure that the other processes had completed) and then continue its execution.

In applications that wish to view executions created with **mounting** as non-terminating, the execution will typically have an initial section that sends a reference for a newly created 'task' stream to mutually agreed upon client streams (by an explicit **posting**). The referenced task stream will then be used to supply the newly mounted execution with additional operations to perform after it completes its starting procedures.

Remote Closures

An value may be sent to a remote site, a reference for it created there, and the reference sent to specified clients using

```
(loading value on site for clients ...)
```

The client streams are returned immediately by the form, and they will eventually receive a reference for the value loaded on the specified site.

A remote closure may be created by

```
(loading #'(lambda arglist form) on site for clients ...)
```

It may then be applied to locally evaluated arguments by passing it those arguments as in

(**passing** *parameter-list* **to** *closure-reference* **for** *clients* ...)

The result of the remote application is sent to the specified clients. The **loading** and **passing** operations are combined in **spawning**.

2.2.6 Miscellaneous Utilities

A few utility operations are provided by LAMINA to specify computation and storage sites, dismiss computations, and provide a timeout facility for applications desiring one. LAMINA also provides simulation control facilities to initiate a simulation, read the current simulation time, and do a computation without increasing the simulation time.

The function **random-site** returns a identifier for a site chosen randomly with uniform distribution over the processor sites in the simulated system. The function **random-memory** does the same thing over the memory controllers in the system. The function **local-site** returns an identifier for the site executing the function. The function **local-memory** returns an identifier for a memory controller associated with the processor on which the function is executed.

In order to provide a timeout facility, the keyword **after** followed by a number of milliseconds in simulated time may be included in functions that take LAMINA keyword arguments. The simplest use might be to specify that a posting to a stream be sent at some future time.

A call to **dismiss** breaks execution. With no argument, execution is rescheduled immediately (but occurs after all previously scheduled executions are run). If an argument is specified which is a non-**nil** symbol, execution is terminated and will never be rescheduled. If a local stream is specified, execution is rescheduled when next that stream receives a posting—or immediately, if that stream has a posting on it.

The current simulation time in milliseconds is returned by the function **simulation-time**.

Some computations in a simulated application need not (or should not) be timed. The macro **without-clock** may be used to wrap such computations so that they are accomplished 'off the clock'. This is generally a good idea for calls to debuggers and the like as well as for input and output operations

Something special must be done to start up a simulation. The form

(**boot** (**at** *time site-coordinates form*) (**at** ...) ...)

will spawn computations to execute forms at the indicated sites beginning at the specified times (in milliseconds). The site coordinates are given as a list, for example, '(3 2), whose length matches the represented dimensionality of the processing unit (a surface for the case shown). The **boot** construct resets the simulator and thus may only be executed as the first operation of an application being simulated. Note that **boot** spawns rather than mounts a computation. If a mounted computation is needed, it must be explicitly mounted by the computation that **boot** spawns.

2.3 Functional Programming

Perhaps the style of computation most readily treated as concurrent is that of functional programming. LAMINA supports concurrent programming using this style by providing means to (1) spawn computations that will provide values to futures and (2) accept such values in a computation—scheduling the computation when they are available. The constructs defining the LAMINA interface for functional programming are

- (**future** *form*) spawns execution of a *lexical closure*, that is, a procedure body to execute a given form together with an environment (determined by the rules of lexical scoping) in which to do the execution [3]. This closure is executed (eagerly) on a randomly selected site. A future which will contain the value of the computation when it is available is immediately returned.
- (**with-values** *future-bindings forms*) spawns an evaluation on the local site to execute the closure corresponding to the *forms*. The evaluation is done within an environment that includes bindings for given variables to the values available for the indicated futures. The evaluation is deferred until all of the indicated futures have values that are not themselves futures. The immediate result of executing a **with-values** form is a future whose value will be supplied by the deferred evaluation.

 Each element of a *future-bindings* list is itself a list: (*binding-pattern future-specifier*). If evaluation of a future specifier in a **with-values** construct produces a value other than a future, the value is encapsulated by a future. After all specified futures have values (which are not themselves futures), the values of each of the futures are *destructured*, that is, the values are treated as list structures and the elements of these list structures are used to bind corresponding variables in a binding pattern of arbitrary depth. These bindings will be included in the environment in which the spawned computation is executed. Only **with-values** can be used in LAMINA to reduce futures to values. Values of futures are never taken as an ancillary consequence of any other operation.

 The results of the evaluation spawned by **with-values** are returned as a future which will receive the value of the spawned computation. The spawned evaluation is treated as the continuation of the spawning computation, and, as such, captures all stack allocated temporary variables required to execute that computation. Thus, each spawned computation may be viewed as running to completion; its continuation, if any, is an independent spawned computation.

All spawned computations run to completion (although they may be suspended by system level operations), and so the stack of the executing processor is generally left clear. Therefore any space allocated for it may be reused by the next computation on that processor, allowing the advantages of stack-based operation without incurring the space penalty discussed in section 2.1. The costs of heap allocation are incurred only as needed.

2.3.1 Ordering: An Example

To illustrate the use of the LAMINA functional programming interface, we develop parallel implementations of the serial (quicksort) algorithm to associate ordering information with sets of numbers, shown in figure 2.2.

The input to `order0` is sets of numbers to be ordered. Elements of a set are the sequential elements of a list, and sets are separated by a `nil` token. The sets (including their separator tokens) are concatenated to form the input list The output is a list with each ordered set represented by successive elements of a list

```
(defun ORDER0 (input-list)
  "Serially order elements of input sets"
  (if (null input-list) ; done
      nil
      (let ((pivot (car input-list)))
        (if (null pivot) ; end of set marker
            (cons nil (order0 (cdr input-list))) ; do next set
            (multiple-value-bind (smaller larger rest)
                ;; partition set around pivot
                (part1 pivot (cdr input-list))
              ;; recur and collect
              (let ((ordered-smaller (order0 smaller))
                    (ordered-larger (order0 larger))
                    (ordered-rest  (order0 rest)))
                (append ordered-smaller
                        (list pivot)
                        ordered-larger
                        ordered-rest)))))))

(defun PART1 (pivot input-list)
  "Partition one set of input around pivot"
  (let ((input (car input-list)))
    (if (null input) ; separator token
        (values () () input-list)
        (multiple-value-bind (smaller-part larger-part rest)
            (part1 pivot (cdr input-list))
          (if (> input pivot)
              (values smaller-part (cons input larger-part) rest)
              (values (cons input smaller-part) larger-part rest))))))
```

Figure 2.2: Serial Ordering

and separated from other ordered sets by `nil` tokens. The sets follow each other in the output in the same order in which they appeared in the input. For example, the input list

```
(7 9 4 nil 5 3 8 nil)
```

would result in the output

```
(4 7 9 nil 3 5 8 nil)
```

Thus the information concerning the ordering of the elements of a set and the identity of that set is implicit in the output.

The result of ordering `nil` is `nil`. If the input list is not `nil`, the first element of that list is used as a `pivot`; otherwise, it is a separator token, and the result then is the separator followed by the result of ordering the rest of the list. A numeric `pivot` is used by `part1` which returns: the (unordered) elements of the current set smaller than the pivot, the (unordered) elements of the current set larger or equal to the pivot, and the remaining elements of the input.

Functional Ordering

A parallel version of `order0` is shown in figure 2.3.[1] The function `order1` recursively spawns itself with each of the three sublists returned by `part1`, then waits for the ordered results. When these are available, it appends the ordered sublist of elements that were smaller than the pivot to the list formed by the pivot, the ordered sublist of elements that were not smaller than the pivot, and the result of ordering the rest of the sets in the input.

The operation of `order1` is characterized by much waiting for the results of spawned computations. The pattern of execution is to spawn a set of computations—using `future`—and immediately wait for all their values to be produced—using `with-values`. This waiting represents serialization due to data dependencies and can significantly limit the concurrency of an algorithm. If, instead, computations can be handed just what they each require to get started (with promises for the rest), they can be pipelined as computation assembly lines, each station operating on a piece of the input from upstream producers and delivering a piece of the output to downstream consumers.

Pipelined Functional Ordering

A pipelined ordering algorithm is developed in figure 2.4. This scheme recursively forms a tree of spawned computations with one leaf for each element and each separator token in the sets of elements to be ordered. Each non-leaf node of the ordering tree partitions its input by sending each input element it receives (from its upstream parent) to one of its two downstream children. The smaller child was created such that its result is used as the result that the parent was asked to produce and the rest of its input is the result of the larger child. The larger child created such that if it is a leaf (that is, if it has nothing to order), its result will be the rest of the items given to the parent. The rest of the items seen by the largest descendent of the

[1] The '?' as the first character of a variable name is purely notational convenience to identify a potential future.

```
(defun ORDER1 (?input)
  "Recursively spawn ordering partitioned input sets"
  (WITH-VALUES ((input-list ?input))
    (if (null input-list)
        ()
        (let ((pivot (car input-list)))
          (if (null pivot)
              (WITH-VALUES ((rest (order1 (cdr input-list))))
                (cons nil rest))
              (multiple-value-bind (smaller larger rest)
                  (part1 pivot (cdr input-list))
                (WITH-VALUES ((ordered-smaller (FUTURE (order1 smaller)))
                              (ordered-larger (FUTURE (order1 larger)))
                              (ordered-rest (FUTURE (order1 rest))))
                  (append ordered-smaller
                          (list pivot)
                          ordered-larger
                          ordered-rest))))))))
```

Figure 2.3: Functional Ordering

smaller child is the result produced by the smallest descendent of the larger child. Thus, using an approach similar to *difference-lists* in logic programming, the results of the leaf elements are tied together to produce the result of the ordering tree.

The first input a child receives establishes the pivot for partitioning, unless it is the separator token, `nil`. If it is `nil` and there is more input, the child returns `nil` as the first part of the result together with a promise for ordering the rest of its input followed by those values larger than anything in that input. If there is no more input, it just returns promises for the results of its larger relatives, that is, the `rest-pair`.

The receipt of a separator token while partitioning indicates that all the elements of a set to be ordered have been received. A terminator, `nil`, is passed to the smaller child and a separator followed by the rest of the unordered input (if any) is passed to the larger child.

The code for this example is written using single-valued futures. Sequences of values are repres nted (that is, 'streams' are simulated) by pairs consisting of a value and a future for the rest of the sequence. The value of the future, when available, is a pair which itself consists of a value for the next element in the sequence and a future for the rest of the sequence. The consequence of this approach is that many short lived dynamic references are created (so that each element of the sequence has an independent reference) and then abandoned. Reclaiming the space allocated for them requires the expensive global synchronization as discussed in section 2.1.

Relaxation of the single value assumption for structures representing unavailable values is discussed in the following section.

```
(defun ORDER2 (?input &optional rest-pair)
  "Future pipeline: rest and input pair (or its future) => ordered pair"
  (WITH-VALUES (((pivot . rest-input) ?input)) ; Coerce value
    (if (null pivot)
        (if (null rest-input)
            rest-pair
            (cons nil (FUTURE (order2 rest-input rest-pair))))
        (WITH-VALUES (((?smaller ?larger)
                       ;; Get promises for first elements of partition
                       (FUTURE (part2 pivot rest-input))))
          ;; Spawn larger ordering, continue ordering smaller
          (order2 ?smaller
                  (cons pivot (FUTURE (order2 ?larger rest-pair))))))))

(defun PART2 (pivot ?input)
  "Produce (<future> <pair>) or (<pair> <future>) for (<smaller> <larger>)"
  (WITH-VALUES ((input-pair ?input)) ; Coerce value
    (if (null input-pair)
        nil
        ;; Destructure pair as (value . future)
        (destructuring-bind (input-value . rest) input-pair
          (if (null input-value)
              (list nil (cons nil rest))
              ;; Spawn continuation of this partitioning
              (let ((part? (FUTURE (part2 pivot rest))))
                ;; and get futures for destructured value of continuation
                (let ((?smaller
                       (WITH-VALUES ((value ?part)) (first value)))
                      (?larger
                       (WITH-VALUES ((value ?part)) (second value))))
                  ;; Return list: (<future> <pair>) or (<pair> <future>)
                  (if (> input-value pivot)
                      (list ?smaller (cons input-value ?larger))
                      (list (cons input-value ?smaller) ?larger)))))))))
```

Figure 2.4: Pipelined Functional Ordering

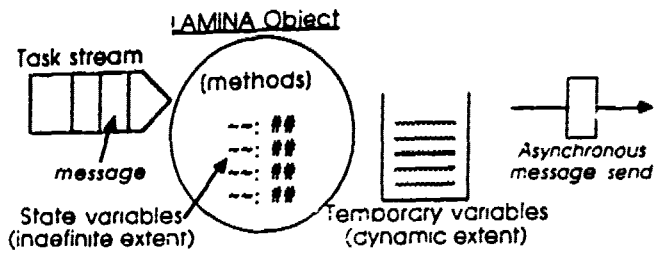


Figure 2.5: Message passing model

2.4 Object Oriented Message Passing

The LAMINA object programming model is founded on the notion of asynchronously communicating objects. An object, as used here, is a collection of variables—its *state variables*—manipulated by (and only by) a set of procedures—the *methods* associated with that object. Objects may be defined within a compiled class inheritance network: the current implementation uses the inheritance facilities of *Flavors*.

A LAMINA object is allocated in *local* space and is referenced externally by its *task stream*, a stream maintained as one of its state variables. The information placed on this stream (that is, provided as its values) specifies *tasks* for the object; each unit of information is called a *message*. A message is internally structured as a *task request posting*, whose value consists of a *task selector* symbol that identifies the method to execute, along with the associated parametric values for the execution.

2.4.1 Computational Flow

As illustrated in figure 2.5, the messages arriving on an object's task stream specify tasks to be performed by that object. Every object has a spawned *dispatch process* associated with it that removes and executes each message on its task stream in turn. Tasks usually mutate the state variables of the object and generate new messages. They have exclusive access to their environment (i.e., state and temporary variables) during execution.

Tasks are *data driven* in that they are started only when all the needed information is available. Typically, a single message, in conjunction with the object's state variables, contains all the relevant information for the task execution. Tasks are generally intended to be accomplished as the stages of pipelines that organize the work performed by the objects of the application. In order not to block the pipeline, a task, once started, is run to completion.

Providing Atomicity

Although LAMINA provides the programmer with a run-to-completion model, there may be system reasons for preempting a task, for example, to handle a debug trap or because the task's run quantum has expired. When this occurs, the object does not execute any other tasks until the preemption is resolved. This prevents

other tasks on that object from gaining access to the environment of the suspended task. However, since other objects may execute tasks during this time, true atomicity can only be enforced if no state is shared between execution environments. The mechanism by which objects communicate ensures this.

LAMINA objects can never share state because they only communicate by exchanging messages containing *independent* copies of local structures. Furthermore, the state variables of an object are only visible to its own methods and are therefore only accessible within a private task. Thus the atomicity of operations on an object is preserved even in the presence of preemptions.

2.4.2 Programming Objects

Sending a Task Request

Sending a task request message in LAMINA is non-blocking so as to directly accomodate pipelined operations on objects. The construct for asynchronously sending a message is `sending`, which takes as arguments one or more target task streams, a task selector symbol, and a list representing the parameters to be provided to the task executions. Since `sending` is no more than syntactic sugar for the `posting` primitive, the sender may provide additional control or debugging information as described in section 2.2.

The value immediately returned by `sending` is the list of clients supplied. As a convention, the *clients* may expect to receive consequent task requests later in the computation.

Defining Objects

LAMINA object types are built upon the base *flavor*, `lamina`, which defines the instance variable, `Self-Stream` that stores its task stream. The default kind of task stream is a normal unordered stream; the 'mixin' flavors `ordered-self-stream` and `sequenced-self-stream`, are provided to override this default.

As an example similar to the one discussed in section 2.3, a LAMINA object to associate ordering information with the numerical values of the elements of sets is defined in figure 2.6. In the example, the state variables of an `order3` *ordering object* are all named, the default initializations specified, and any state variables to be initialized by a creator are identified.

Trigger Methods

The 'top level' methods executed as tasks by LAMINA objects are called *triggers*. They are defined using the `deftrigger` macro as shown in figure 2.6. The parameter list provided to `deftrigger` corresponds to the value (and the other information, which can be optionally ignored) containined in each `posting` received on its task stream. In particular, the parameter specification may be used to destructure the value provided, as is done in the example.

Creating Objects

The form

(`creating` *type initializations* `for` *clients* `on` *site* ...)

stipulates the creation of a object on the indicated site, or on a randomly selected site if none is indicated. When the creation has been accomplished, the client streams will receive a posting whose value is the task stream of the created object.

The *initializations* are a list of alternating keywords (corresponding to the state variable names for the object being created) with their initial values. These values are computed in the context of the object requesting creation. As an example, `creating` forms are included in the `order3 :control` method definition shown in figure 2.6. The function `create-self-stream` is provided to create a stream as defined by the type of the LAMINA object being created. This can be used, as in the example, to create the object's task stream before the actual object has being created.

2.4.3 Ordering Example

In figure 2.6, iteration and assignment replace the recursion and binding used for the functional programming ordering example shown in figure 2.4. Sequences of values on streams are represented by long lived streams that couple producing and consuming ordering objects. The objects make use of supporting procedures defined in figure 2.7.

Each `:element` message manipulated by the ordering routine indicates the value of the element to be ordered and the set in which that element appears. The output `:element` messages include this information together with the calculated order of the element in the indicated set. An `:end` message may be generated either by the root calculation requesting a set be ordered or by intermediate ordering objects serving that calculation. Each such message includes a set identifier, the number of elements the receiver should expect for that set, and the (`base`) order of the smallest element to be expected. The `order3` objects keep track of this information for each set they are dealing with via a property list of `control` records. The set of an input is used to retrieve the appropriate control record from among those in use by the object.

If there is no pivot yet received to use in partitioning the set, the ordering object saves the input value as the pivot for the set. Otherwise, the `:element` trigger method passes the input element to either its larger or smaller child and counts the number of elements sent to the smaller child. If all the expected inputs for a set have been received, an `:element` message including the value, the set, and the order of the value in the set will be sent to the result stream. An `:end` message will be sent to any children that have been sent elements of the set to order.

2.4.4 Implicit Continuations

For LAMINA objects, continuations of a computation are often some explicit trigger method of some explicit object. There are cases, however, in which it is inconvenient to create an explicit name for a continuation. As

```
(defstruct CONTROL
  ((pivot nil) (base nil) (expected nil) (count 0) (smaller 0)))

(defflavor ORDER3
           ((Controls '())
            (Smaller-Child nil)
            (Larger-Child nil)
            Result-Stream)
           (LAMINA)        ; inheritance
  (:initable-instance-variables Result-Stream))

;;;
;;; 'Trigger' methods
;;;
(DEFTRIGGER (order3 :ELEMENT) (input)
  "Partition by established pivot or get pivot; check for completed set"
  (let* ((value   (first input))
         (set-id  (second input))
         (control (send self :control set-id)))
    (cond
      ((null (control-pivot control)) ; set pivot for [new] set
       (setf (control-pivot control) value))
      ((>= value pivot)
       (SENDING Larger-Child :element input))
      (:else
       (SENDING Smaller-Child :element input)
       (incf (control-smaller control))))        ; Count smaller in set
    (send self :completed? control set-id)))

(DEFTRIGGER (order3 :END) ((base set-id expected))
  "Note base and send :end to children if complete"
  (let ((control (send self :control set-id)))
    (setf (control-expected control) (1+ expected))
    (setf (control-base control) base)
    (send self :completed? control set-id)))
```

Figure 2.6: Ordering with LAMINA Objects

```
;;;
;;; 'Regular' methods
;;;
(defmethod (order3 :control) (set-id)
  "Get or create control for input; maybe make descendants"
  (unless Smaller-Child
    (setf Smaller-Child (CREATE-SELF-STREAM 'order3)
          Larger-Child  (CREATE-SELF-STREAM 'order3))
    (CREATING 'order3 (list :self-stream Smaller-Child
                            :result-stream Result-Stream))
    (CREATING 'order3 (list :self-stream Larger-Child
                            :result-stream Result-Stream)))
   (or (getf Controls set-id)
       (setf (getf Controls set-id) (make-control)))))

(defmethod (order3 :completed?) (control set-id)
  "Count received in set against expected and finish set if complete"
  (let ((expected (control-expected control)))
    (when (eql expected (incf (control-count control)))
      (let ((pivot (control-pivot control))
            (base (control-base control))
            (smaller (control-smaller control)))
        (let ((pivot-order (+ base smaller))
              (larger (- expected smaller 1)))
          (SENDING Result-Stream :element (list pivot set-id pivot-order))
          (when (plusp smaller)
            (SENDING Smaller-Child :end (list base set-id smaller)))
          (when (plusp larger)
            (SENDING Larger-Child :end (list (1+ pivot-order) set-id larger)))
          (remf Controls set-id))))))
```

Figure 2.7: Support for Ordering Objects

```
(DEFTRIGGER (distributer :start-servers) ((count input-stream))
  "Round robin distribution of input requests to created server pairs"
  (let ((servers nil)
        (as (CREATING 'a '() for (NEW-STREAM) on (RANDOM-SITES count)))
        (bs (CREATING 'b '() for (NEW-STREAM) on (RANDOM-SITES count))))
    (WITH-POSTINGS ((a as) (b bs))
      (cond
        ((null servers) ; first invocation of continuation
         (setf servers (circular-list (list a b))) ; single elt
         (WITH-POSTINGS ((request input-stream)) ; start distributer
           (SENDING (pop servers) :request request))) ; multicast
        (:else          ; other invocations, upto count
         (push (list a b) (cdr servers)))))))
```

Figure 2.8: Implicit Continuations

a syntactic construct, execution of a continuation of a computation can be specified to occur in the context of an executing object (as defined by its set of state variables and the environment of the continuation) each time that postings have been received on some given streams. The execution spawning the continuation is finished normally and then the next operation to be done on the object is taken from its task stream without delay. Thus LAMINA objects can be viewed as *monitors* (because the independently atomic operations on objects give the required mutual exclusion) but operations on them are unnested. This is done to facilitate pipelined operation: task request postings queued for operation on an object are not deferred for a pending continuation.

The construct

(with-postings *stream-bindings form*)

creates an implicit continuation in the context of an object. The *stream-bindings* is a list, each element of which is a list of a *binding-pattern* and a *stream*. Each of the postings on the indicated streams (including the posting clients, tag, key, origin, and properties) will be destructured and bound to a corresponding variable identifier according to the associated *binding-pattern*. These variables and associated values are also part of the execution environment of the continuation.

As an example of the use of **with-postings**, consider the example shown in figure 2.8. It uses nested **with-postings** constructs to create continuation closures that first create and collect pairs of `lamina` objects and then distribute requests on an input stream to the collected triples in a round robin fashion. Note that instance variables may be accessed by the continuations.

The implicit continuation will be executed atomically with respect to any other operations on the indicated object and in the context of its state variables and the lexical environment in which the form appears. A schematic of the mechanism supporting implict continuations in objects is shown in figure 2.9.

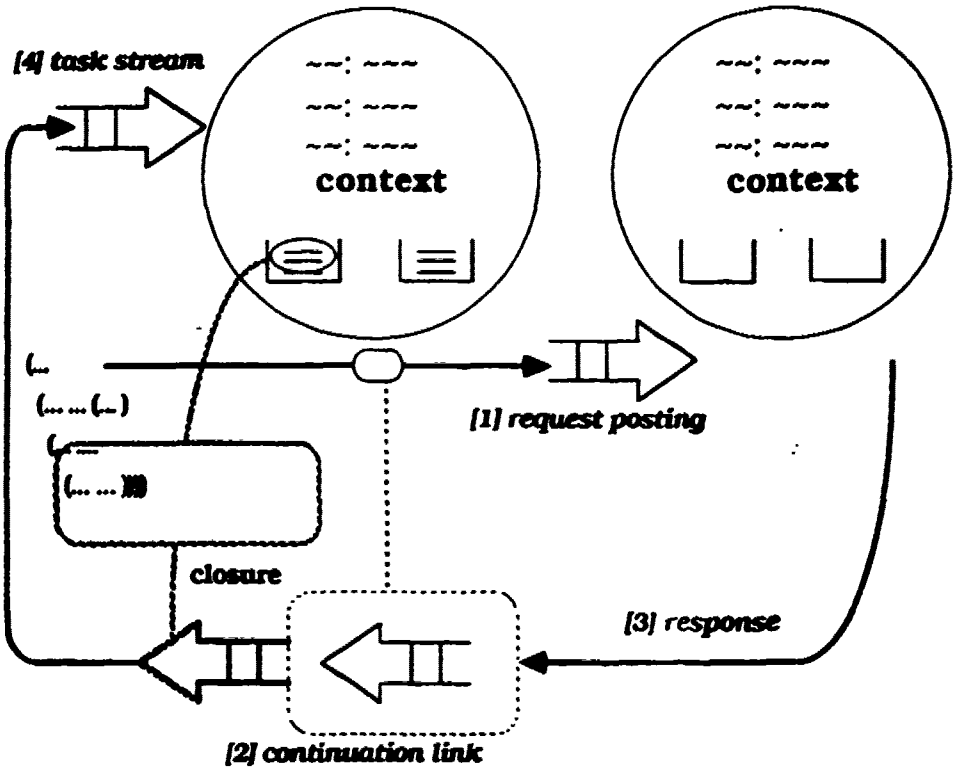


Figure 2.9: Continuation Closures

[1] References for streams on which responses are expected are sent in task request postings to other objects as places to supply response postings. [2] Intermediate variables (that is, the environment) and a pointer to a block of code required to execute the form wrapped in a **with-postings** construct are captured in a continuation closure, attached to a stream, and linked to the stream(s) on which responses are expected. [3] When all required postings become available on these streams, [4] the response postings together with the closure are sent to the task stream of the object that generated the closure.

The closure is executed (in its turn) atomically within the context of the object and lexical environment of the form. Variable bindings are made as specified to the elements of the available response postings. Note that the execution that spawned execution of the closure and the execution so spawned are independently atomic. The state variables of the object and any structures they reference can be changed by some other operation taken from the task stream between the two executions. The syntactic convenience is only that: invariants that need to be preserved across independent executions need to be met at the boundaries between the execution that spawned execution of the closure and the execution so spawned.

2.4.5 LAMINA Objects and ACTORS

The LAMINA object model is similar to ACTORS, in that message arrival triggers computation and message arrival order is non-deterministic. However, it departs from ACTORS in a number of ways, primarily by trading off flexibility for efficiency.

- Not everything is an object. Predefined data types such as numbers, symbols, arrays and cons cells exist as primitives, and operations on them do not entail message-passing. Although structures are passed by copying, they are locally mutable.
- Streams are first-class entities independent of objects. Objects may establish communications over streams other than their task streams. Streams may also be shared between objects as described in section 2.2.
- The default operation of a LAMINA object is serial command execution. For serial execution sequences, stack allocation of dynamically allocated structures can be used where the compiler can determine that references to the structures have dynamic extent.
- Mutation is explicit. Unlike actors, LAMINA objects do not deal with state changes by specifying a *replacement* actor for themselves, but rather explicitly manipulate their own state variables through assignment.
- Although structures are passed by copying, they are locally mutable. Tasks may change them and pass the changed structure to some other object. The copying done to transmit the structure will occur asynchronously with method execution.
- Finally, LAMINA relies on compiled inheritance for method combination rather than upon runtime delegation, and it does instantiation by compiled template rather than by copying a selected instance with specified exceptions.

2.5 Shared Variables

Shared variables in LAMINA are cells that are managed by a memory controller and whose associated value may be mutated. LAMINA also supports shared data pairs ('conses') and arrays. A shared variable reference is constructed, accessed, and mutated by the interface operations described in this section. For all these operations, execution is deferred and no other executions are performed by the initiating processor until the indicated operation is accomplished.[2]

Shared queues (which are modelled using streams) are also provided. These queues are maintained in a processor's local memory. When a process reads from a shared queue, it is halted and descheduled; execution is resumed when the requested data arrives.

[2]Note that, because the simulator is executing in a uniprocessor environment, a stack must be maintained for each deferred execution. Thus executions must be resumable (not merely restartable) to use the shared variable LAMINA interface. This is discussed in section 2.2.

2.5.1 Creating and Accessing Shared Variables

A single shared variable can be allocated and initialized using the `shared-variable` operator, that takes as a required argument the initial value for the shared variable, and creates and returns a reference to a cell containing the indicated value. The value of the cell, in general, must be a self-referential datum or a dynamic or static reference.

An optional argument can be used to specify a memory site at which to allocate the cell; if it is omitted, a randomly selected memory site is chosen. Alternatively, the macro

(`in-memory` *site-identifier* ...)

can be used to specify a default site for all allocations (for simple as well as structured shared variables) performed within its dynamic scope.

Once a shared variable has been allocated, the following constructs may be used to access or alter its value:

- (`shared-read` *cell*) retrieves the value of the referenced cell.
- (`shared-write` *cell value*) modifies the value of the referenced cell. The new value is returned.
- (`shared-exchange` *cell value*) performs the same function as `shared-write`, except that prior value of the reference is returned. The exchange is atomic.
- (`shared-replace-conditional` *cell old new*) atomically compares the contents of the referenced cell with *old*, and, if they are identical, replaces the contents with *new*.

For each of these constructs, the operation is guaranteed to be completed before execution is resumed.

2.5.2 Shared Data Structures

LAMINA also provides support for shared data structures, namely shared pairs and shared arrays. Shared pairs form the foundation of linked data structures such as lists and graphs.

The constructor

(`shared-cons` *car-value cdr value*)

creates and initializes a shared pair, returning a reference to it. The accessors are, naturally, `shared-car` and `shared-cdr`, while the mutators are `shared-rplaca` and `shared-rplacd`. Also, the form

(`cache-shared-pair` *shared-pair-reference*)

may be used to make a local, that is, non-shared, copy of a shared pair in local space

The form

(**shared-array** *dimensions*)

returns a reference to a shared array. The *dimensions* argument is a list of positive integers, denoting the size of each dimension of the array. There are optional **:initial-element** and **:initial-contents** keyword arguments, which may be used, respectively, to initialize all the elements of the array to the single value specified or to initialize each element of the array to the value of the corresponding element in a list or a list of lists. Shared arrays are initialized to **nil** by default.

The accessor **shared-aref** reads elements of the shared array. The mutator **shared-aset** writes array elements. Both operations are bounds-checked against the dimensions of the array. Finally, the **cache-shared-array** function returns a local (non-shared) copy of the referenced shared array, while **fill-shared-array** copies data from a local array into a shared array.

2.5.3 Shared Queues

A shared queue construct, which is implemented as a LAMINA stream, is also provided. Shared queues are managed by a processor which provides atomic access to the queue and, when the queue is empty, maintains a FIFO queue of processes requesting data from it; the requests are serviced when data is added to the queue. Further, whenever a process attempts to remove data from the queue, the process is descheduled; execution is rescheduled when the requested data arrives.

Shared queues are created by the **shared-queue** function, which takes one optional argument representing the queue's tag, which may be used for debugging. Items may be added to the queue with the **shared-enqueue** function. The **shared-dequeue** function removes and returns the top item of the queue, while the **shared-queue-top** function merely returns it.[3] A **shared-queue-p** predicate is also provided to test whether an item is a shared queue.

Unlike other shared variable operations, accesses to shared queues do not cause the initiating processor to stall waiting for completion. A process executing **shared-queue** continues immediately, without waiting for the data to arrive on the queue. A process which accesses a queue, using **shared-dequeue** or **shared-queue-top**, will be halted and descheduled. Execution is rescheduled when the data arrives, but the initiating processor may perform other executions in the meantime.

2.5.4 Other Synchronization

A simple spin lock is provided for busy-wait synchronization. A lock is implemented as a cell that is initialized to a value other than **nil**, and and the atomic exchange operation is used to set and clear it. The form

(**with-spin-lock** *lock form*)

[3] In the current implementation, only FIFO queues are provided, and (in order to maintain a consistent timing model for cross address space transmissions) only shared variable or shared queue references may be placed on a shared queue.

executes the given form after aquiring the referenced lock; subsequently, the lock is released and the value produced by the execution of the form is returned.

Such a synchronization operator might be used in incrementing a shared counter as in[4]

```
(defun locked-increment (<counter> <lock> &optional (delta 1))
  (WITH-SPIN-LOCK <lock>
    (SHARED-WRITE <counter> (+ (SHARED-READ <counter>) delta))))
```

Locks can also be constructed from shared queues, as is done by LAMINA to implement mutual exclusion locks. To release the lock, a process places a token reference on the queue. A process acquires the lock by removing the token—any other process which attempts to remove it will be blocked until the owner of the lock replaces the token. Alternatively, reading but not removing the token (by using `shared-queue-top`) allows more than one process to be resumed. This last approach more closely resembles the type of synchronization provided by signalling and waiting on condition variables in a monitor.

Figure 2.10 shows an example of using some of these synchronization schemes in generating a closure to perform operations on a shared buffer realized as a shared variable array.[5] Processes first gain access to the shared array by spinning on a lock. Once access is granted, items are inserted or removed. An attempt to put information in a full buffer returns `nil`. When an attempt is made to remove data from an empty buffer, a shared queue (rather than data) is returned; the requesting process may then wait for something to be placed on this queue by executing `shared-queue-top`.

2.5.5 Ordering Example

As an example of using the LAMINA shared variable interface, we present yet another implementation of ordering, this one using shared variables. The sets to be ordered are represented as shared arrays.

Each processor executes an identical *thread of execution*, as defined by the `order4` function that is shown in figure 2.12. Ordering requests are distributed to the threads through a shared buffer manipulated by a closure previously formed by calling the `shared-buffer` function. A request consists of a reference to a shared array and indices representing the left and right boundaries of the array (or sub-array) to be ordered. Each thread executes in a loop as follows:

- If there is an array (or sub-array) to order, the thread partitions the sub-array, using the `part4` routine, shown in figure 2.11. The order of the set element used as the pivot is now established so the set element, its order, and the reference for the array (as a set identifier) is placed in the specified result queue.
- If both sub-arrays resulting from the partition are longer than two elements, the thread adds an ordering request to the queue for one sub-array and orders the other. If either sub-array has two or fewer elements, the ordering is trivial, so the thread does it (using the `maybe-exchange` function, also shown in figure 2.11) If neither sub-array has more than two elements, after the thread orders the

[4] By convention, references to shared variables and shared queues are denoted by enclosing angle brackets, as in `<lock>`.

[5] The astute reader will note that the closure environment itself is not explicitly represented as shared; this is a modelling convenience due to the fact that the environment is not modified during the lifetime of the closure.

```
(defun SHARED-BUFFER (size)
  (let ((<signal> (SHARED-QUEUE ':signal))
        (<empty>  (SHARED-VARIABLE t))
        (<lock>   (SHARED-VARIABLE t))
        (<buffer> (SHARED-ARRAY size :initial-element nil))
        (<head>   (SHARED-VARIABLE 0))
        (<tail>   (SHARED-VARIABLE 0)))
    #'(lambda (operation &optional value)
        (WITH-SPIN-LOCK <lock>
          (let* ((head (SHARED-READ <head>))
                 (tail (SHARED-READ <tail>)))
            (ecase operation
              (:insert
               (let ((new-tail (mod (1+ tail) size)))
                 (if (= head new-tail)
                     nil
                     (prog1
                       t
                       (SHARED-ASET value <buffer> tail)
                       (when (SHARED-READ <empty>)
                         (SHARED-WRITE <empty> nil)
                         (SHARED-ENQUEUE <signal> <signal>))
                       (SHARED-WRITE <tail> new-tail)))))
              (:remove
               (if (not (= head tail))
                   (let ((new-head (mod (1+ head) size)))
                     (SHARED-WRITE <head> new-head)
                     (SHARED-AREF <buffer> head))
                   (unless (SHARED-READ <empty>)
                     (SHARED-WRITE <empty> t)
                     (SHARED-DEQUEUE <signal>)))))))))))
```

Figure 2.10: Shared Buffer

```
(defun PART4 (<array> first last)
  "Does partition on array, and returns position of pivot"
  (labels
    ((first-larger (<array> index limit pivot)
       (loop for position from index to limit
             as item = (SHARED-AREF <array> position)
             when (>= item pivot) return (values item position)
             finally (return (values item (1- position)))))
     (first-smaller (<array> index limit pivot)
       (loop for position downfrom index to limit
             as item = (SHARED-AREF <array> position)
             when (<= item pivot) return (values item position)
             finally (return (values item (1+ position)))))
     (part4-step (<array> left right pivot pivot-index)
       (multiple-value-bind (larger-item larger-index)
           (first-larger <array> (1+ left) right pivot)
         (multiple-value-bind (smaller-item smaller-index)
             (first-smaller <array> right left pivot)
           (cond
             ((> smaller-index larger-index)
              (SHARED-ASET smaller-item <array> larger-index)
              (SHARED-ASET larger-item <array> smaller-index)
              (part4-step
                <array> larger-index (1- smaller-index) pivot pivot-index))
             (:else
              (SHARED-ASET smaller-item <array> pivot-index)
              (SHARED-ASET pivot <array> smaller-index)
              smaller-index)))))))
    (part4-step <array> first last (SHARED-AREF <array> first) first))

(defun MAYBE-EXCHANGE (<array> first second)
  "Exchanges first and second items, iff first is greater."
  (let ((first-item (SHARED-AREF <array> first))
        (second-item (SHARED-AREF <array> second)))
    (when (> first-item second-item)
      (SHARED-ASET second-item <array> first)
      (SHARED-ASET first-item <array> second))))
```

Figure 2.11: Shared Variable Partition and Exchange

```
(defun ORDER4 (<threads> <lock> requests results &optional request)
  (destructuring-bind (<array> first last) request
    (if <array>
        (let* ((pivot-position (part4 <array> first last))
               (contents (list (SHARED-AREF <array> pivot-position)
                               pivot-position <array>)))
          (funcall        ; Order of pivot data element is established
            results :insert (SHARED-ARRAY 3 :initial-contents contents))
          (let ((left-diff (- pivot-position first))
                (right-diff (- last pivot-position)))
            (cond
              ((and (> left-diff 2) (> right-diff 2)) ; Order right partition
               (let* ((request (list <array> first (1- pivot-position)))
                      (request-block (SHARED-ARRAY 3 :initial-contents request)))
                 (when (null (funcall requests :insert request-block))
                   (order4 <threads> <lock> requests results request))
                 (order4 <threads> <lock> requests results
                         (list <array> (1+ pivot-position) last))))
              ((> left-diff 2) ; Exchange right and then order left
               (when (= right-diff 2) (maybe-exchange <array> (1- last) last))
               (order4 <threads> <lock> requests results
                       (list <array> first (1- pivot-position))))
              ((> right-diff 2)  ; Exchange left and then order right
               (when (= left-diff 2) (maybe-exchange <array> first (1+ first)))
               (order4 <threads> <lock> requests results
                       (list <array> (1+ pivot-position) last)))
              (:else          ; Order by exhange for both left and right
               (when (= right-diff 2) (maybe-exchange <array> (1- last) last))
               (when (= left-diff 2) (maybe-exchange <array> first (1+ first)))
               ;; Declare completion of ordering request and try again
               (locked-increment <threads> <lock> -1)
               (order4 <threads> <lock> requests results)))))
        ;; else get next request
        (let ((<request> (funcall requests :remove)))
          (if (SHARED-QUEUE-P <request>) ; If buffer was empty...
              (if (zerop (SHARED-READ <threads>))  ; signal termination
                  (SHARED-ENQUEUE <request> <request>)
                  (progn (SHARED-QUEUE-TOP <request>) ; or block till signalled
                         (order4 <threads> <lock> requests results)))
              (progn (locked-increment <threads> <lock>)  ; Else, pick up request
                     (let ((request (coerce (CACHE-SHARED-ARRAY <request>) 'list)))
                       (order4 <threads> <lock> requests results request))))))))
```

Figure 2.12: Shared Variable Ordering

sub-arrays, it signals that one less thread is currently working on any ordering requests and notes that it has no array to order.

- If the thread has no array to order, it attempts to remove a request from the queue. If successful, it signals that one more thread is trying to do ordering and orders the (sub-)array identified by the request. If the attempt is unsucessful and there are no other working threads, there will never be any more requests generated so the thread terminates. Otherwise, it tries again to remove a request from the queue. Note that the first thread to terminate places a token on the shared synchronization queue—this wakes up the other threads, which will then terminate.

Chapter 3

CARE System Simulations

This chapter describes the CARE value passing machine model developed to support applications written using LAMINA. In particular, it describes the system components in the CARE library, the parametric variations that may be made to these, and the runtime view of the application presented by the default instrumentation of a configured design. It concludes by detailing the steps that a user follows to simulate a design running an application written using LAMINA.

3.1 A Value Passing Machine Model

In order to simulate the execution of programs expressed using the LAMINA programming model, the hardware system supporting the model needs to be considered. CARE has a component library to provide such support. This library includes the following components:

Evaluators : which are responsible for execution of runnable application processes;

Operators : which are responsible for creating and accepting messages (including encoding and decoding values as necessary) and queueing runnable processes for the evaluator's attention;

Fifo-Buffers : which maintain queues of messages between an operator and the network;

Net-Outputs : which are responsible for transmitting messages from a site to a neighboring site or, as a special case, to the operator's incoming fifo-buffer;

Net-Inputs : which receive messages from neighboring sites and request connections to suitable net-outputs for retransmission to that net-output. As a special case, one net-input on each site receives messages from the operator's outgoing fifo-buffer rather than from a neighboring site;

Sites : which coordinate connections from net-inputs to net-outputs. Site library components have substructure as shown in figure 3.1.

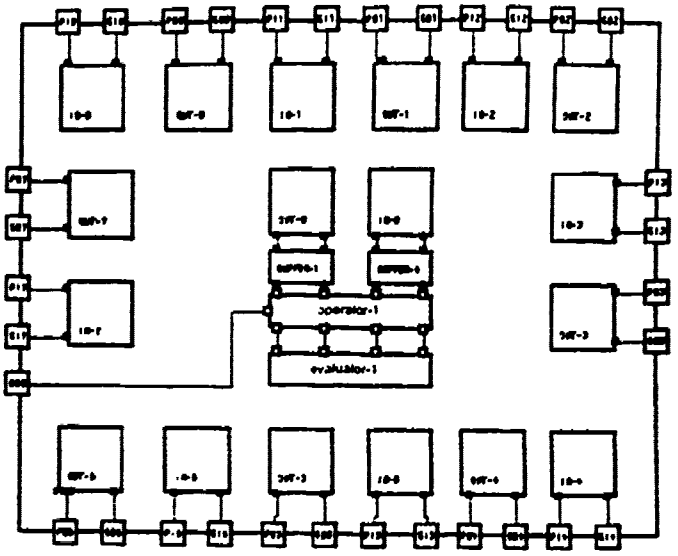


Figure 3.1: Site Library Component

The operator, the evaluator, and the fifo-buffers associated with a given site are assumed to share access to a common local memory. In the value passing machine model, there is no global memory: all memory in the simulated design is local to some site.

Message transmission is accomplished by local flow control and cut-through routing with routing decisions locally and dynamically accomplished through the interaction of a site with an associated net-input (independently of the operator or evaluator at that site). Parametric choices permit modelling of a variety of routing strategies to select among appropriate and available net-outputs as well as variation between cut-through routing and (simple) wormhole routing. In cut-through routing, messages which can not be routed from the site via an appropriate net-output are instead locally buffered by being sent to the operator. The message will then be sent on its way again by the operator at some future time. Multicast transmission with deadlock recovery is supported by the CARE library components by default but may be surpressed if desired.

3.1.1 CARE Library Component Parameters

The operating characteristics of the CARE component models may be changed parametrically to represent a range of alternative hardware systems—both for a value passing model supporting the LAMINA object oriented paradigm and for a reference passing model supporting the LAMINA shared variable paradigm discussed in the next chapter.

The base system cycle time is normally set as 100ns. In this time, each site (asynchronously with other sites) issues one instruction, executes one arithmetic unit operation, makes routing decisions, and transfers

information from the net-outputs of one site to the net-inputs of neighboring sites.

The default settings for the (network and operator) communication facilities of the system reflect expectations that:

- creating and accepting a message each require a minimum of 100 cycles for 'basic service overhead' (to gain the attention of the operator and setup internal registers) no matter how small the message;
- encoding and decoding each require an additional 16 cycles for each 32-bit word transmitted and received—this is the 'packet formatting' time;
- 'transmission' of one 32-bit word from a net-input of one site to the net-input of a directly connected site requires 16 cycles—8 cycles from the net-input to a selected net-output on the site and 8 cycles from that net-output to the net-input of another site directly connected to it (corresponding to a 4-bit data path between sites);
- routing decisions never result in moving information away from the target of that information but they do permit alternative paths to the target. This is labelled 'directed' routing;
- 'queue insertion'—including a message in a stream other than as the last item of that stream (as specified by priority information in the message) requires 10 cycles for each previously included message between the place the new message is put and the last item of the stream;
- 'process creation' (for the value passing model) and 'stack group creation' (for the reference passing model) require 150 cycles to form the appropriate control structures and pre-allocate storage; and
- there is no 'buffer bound' on the fifo-buffers. It is assumed that the shared storage at the site is utilized for this purpose and that this is significantly larger than what is required for message buffering. The user can study performance of systems for which this is not true by setting the fifo-buffer bound to the number of messages that are to be buffered.

The evaluator by default is expected to execute ten million instructions per second. This applies the most stress to the remaining components of a CARE multiprocessor design that a machine with a 100ns instruction issue rate could possibly accomplish. However, the evaluator 'speedup factor' which establishes evaluator performance can be adjusted to model whatever applications performance a single simulated processor is expected to provide relative to the performance of the processor on which the simulation system is being run. For debugging purposes, the time of all evaluator operations can be fixed by setting the 'evaluator override' to a non-null value.

The remaining parameters reflecting evaluator performance are 'context switch override' time (for the value passing model) and 'stack group override' time (for the reference passing model). Both of these are set by default to 300 cycles—representing the time to save and load register contents in a machine and otherwise deal with ("lightweight") process transitions in order to perform an evaluation for the next runnable process. No optimization is made for the case where two successive process executions involve operations on the same context or stack group. The user can use the simulating machine's actual context or stack group switch time by supplying a null value for this parameter.

By default, there is no bound on the number of runnable processes that may be queued for execution by a given evaluator so the 'process queue bound' is null.

The remaining CARE simulation parameters control the documentation and operation of simulation runs. The user may include arbitrary 'comments' in the output of the simulator in order to label results with experimental conditions beyond the simulator parameters just discussed. Lastly, if a simulation is a 'production CARE run', garbage collection and external interrupt conditions are set to improve the repeatability of runs at the cost of some interactiveness in the simulation. When debugging applications, the user should set this parameter to 'no'.

3.2 Seeing Multiprocessor Application Activity

In the LAMINA object computational model, objects interact by passing messages. In the CARE value passing machine model, message arrival triggers component activity. To provide a consistent perspective for the computational model, the machine model, and a measurement model, our measurement model is based on monitoring message traffic. This leads us to an instrumentation system that monitors machine activity in terms that are relevant to the computations performed by the application. We measure message volume, message patterns in space, and message patterns in time to characterize the operation of the computational model. To explain these measurements, we measure network conflicts, scheduling overhead, and synchronization delays in monitoring the operation of the machine model.

By monitoring communications, we monitor process interactions. By monitoring process interactions, we monitor the computation. In order that our simulation is responsive as well as accurate, we simulate only the behavior in response to communication events and the time between such events. No other activity is relevant to our measurements and therefore no other activity is relevant to our simulation. Further, to know the time between communication events driven by execution of the application, we need only time actual execution between communication events. Such activity need not be simulated.

The ability to monitor CARE components and LAMINA objects is provided by means invisible to the applications programmer. In writing applications, no attention need be given to instrumentation for monitoring its activity. The underlying class behaviors of LAMINA objects and CARE components provide this facility.

3.2.1 Panels of the OBSERVER Instrument

When a SIMPLE simulation is run, an *instrument* may be selected for displaying the operation of the model. One option for CARE is the **Observer** instrument. This presents a picture composed of panels listed below—each showing ongoing activity of the simulated design according to a particular perspective.

Network-Operator Map : an animation of the multiprocessor design showing the communication channels currently passing (or attempting to pass) application information between sites and an indication of the number of messages queued for processing by each site's operator.

Processor Utilization : a pair of histograms showing how many of the evaluators and operators in a multiprocessor design have been utilized for what part of the duration of application execution. This panel also indicates the number of currently active evaluators and operators.

Network Load and Latency . a strip chart with a horizontal axis (showing simulation time a message arrives at its target operator) and two vertical axes. One vertical axis shows the latency experienced by messages from the time they are launched into the network by an operator until they arrive at the operator they target—and, as an increment to this time, the time between their creation by an evaluator and their launching into the net by the associated operator. The second axis shows the 'potential' left in the network. This is defined as the number of operators that are not sending a message.

Operator Load and Latency : another strip chart, also with a horizontal axis (showing the simulation time a message is serviced) and two vertical axes. One of these vertical axes shows the time each

message required for service by its target operator—and, as an increment, the time spent waiting for such service. The second axis, as above, shows the 'potential' remaining in the design being simulated. In this case, operator potential is shown. There are two measures of operator potential provided: one for the number of operators with no messages to handle (labelled on the instrument as 'less than one message') and one for the number of operators with less than three messages to handle.

Evaluator Load and Latency : the third strip chart of the **Observer** is much like the second. In this case, the times shown against simulation time are those required to perform an evaluation for an application and the time spent waiting for evaluation after becoming runnable. The potentials shown are those indicating the number of evaluators with less than one and less than three runnable processes.

Cumulative Latencies : message and process execution latencies are presented for each application context in the simulation showing (cumulatively) the time required to launch a message, to transmit it through the network, to wait for the target operator and service the message there, to wait for the target evaluator, and, finally, to run the application code stipulated by the message. This information is ordered from left to right according to which application contexts were associated with the longest cumulative delays until execution was begun.

Activity by Instance : the activity of each object instance receiving a message is presented in the form of 'scrolling text' sorted so that those messages that have experienced the most delay and those objects with the longest expected service time are shown first. The expected service time for an object is computed as the product of the number of messages in the object's task stream and the average time for the object's past computations. These three measures are reported from left to right followed by the number of messages handled by the object, the most recent delay experienced from the time a message was created until the execution it requested was begun, the object's site, and an identification for the object in terms of the kind of object it is, its site, and the simulation time at which it was created.

Activity by Class : the activity of all objects belonging to each class is aggregated and shown in the form of scrolling text sorted so that those classes with the longest average expected service times are shown first. From left to right the information shown for each object class is the average expected service time, average number of messages in task streams, average time to service a message, the total number of messages serviced, the number of instantiated objects in the class, and, finally, an identifier for the class.

Notes : finally, the experimental conditions for the simulation run are summarized. This includes any comments provided by the user when setting up the simulation parameters.

A completed **Observer** instrument is show in figure 3.2.

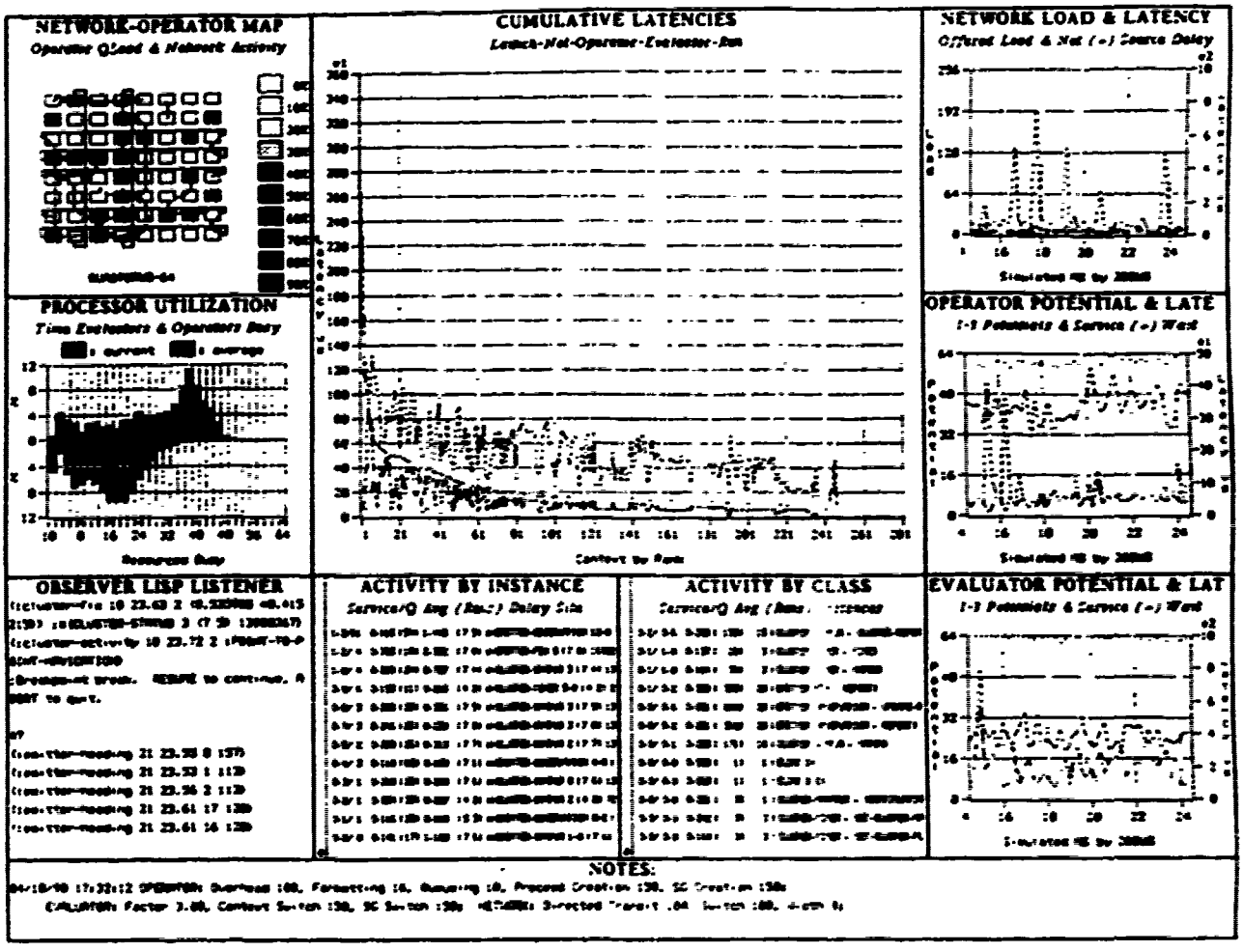


Figure 3.2: Observer Instrument

3.3 Running a CARE Simulation

This section describes the mechanics of simulating a CARE design with an application written in LAMINA.

3.3.1 Loading System Code

The first step in running a simulation is loading the CARE system code into the LISP environment. This is accomplished by executing the following forms:

```
(make-system 'simple-care :noconfirm :silent :nowarn)
(make-system 'care :noconfirm :silent :nowarn)
```

On machines that have been booted with a core image (an *Explorer band*) containing CARE, this is unnecessary.

3.3.2 Setting The Package

The next step is to set the current *package* to one appropriate for using CARE. This is the `care-user` package, within which all the user interface functions are defined. Since the `care-user` package has the nickname `cu`, this can be done by executing:

```
(in-package 'cu).
```

3.3.3 Designs and Instruments

A *design* is a collection of CARE library components that are connected together to represent a multiprocessor. The structure of a design is defined by a graphical editor and is saved in a *design file*. All predefined CARE design files reside in the *care:designs;* directory.

Before a design can be used in a simulation it must be *loaded*. At that time, all of its components are created and are connected according to the saved structure definition.

An *instrument* is a LISP window that displays the activity of a design during a simulation. The instrument window is divided into regions called *panels*. The previous section described the information presented by the panels of the `Observer` instrument.

3.3.4 The simple Function

Most simulation activities are started by calling the `simple` function. This function takes several keyword arguments which allow it to load designs, create instruments, and execute applications. A few of the more important arguments are described below.

- `:design` A symbol or pathname string that specifies which design to use for the simulation. It is first loaded from a design file, if not previously loaded If this argument not present, the most recently loaded design is used. The call:

  ```
  (send s:*box* :name)
  ```

 may be used to check the name of the "current" design.

- `:reset` A value of `t` causes the current design and instrument to be reset (i.e. to move back to the beginning of simulated time and reinitialize).

- `:flush` A value of `t` causes all initial simulation events to be flushed from the event queue.

- `:instrument` This is a symbol that specifies the flavor of instrument to be used for this simulation (for example, `'observer`). If this is not specified, the most recently instantiated instrument for the design is reused. Instead of a symbol, an instrument instance may also be used (for example, from another design; the function `instrument` may be used to retrieve such an instance).

- `:new-instrument` A value of `t` guarantees that a new instrument will be instantiated, rather than an old one being reused.

- `:run` A value of `t` (the default) will cause a fresh simulation to be initiated after the above activities have occurred.

3.3.5 An Example Simulation

The following steps will allow you to run a sample program under CARE:

1. Make sure CARE is loaded as described above.

2. Load the CAMINA code of the sample program, called `LineSim`, which models the voltage transmission across a group of VLSI wires. The implementation reflects an explicit solution to the difference equations representing the voltage characteristics.

   ```
   (make-system 'linesim :silent :nowarn :noconfirm).
   ```

3. Use `simple` to load a design (sixteen sites configured as a torus) and create an instance of the `observer` instrument.

   ```
   (simple :flush t :reset t :instrument 'observer :design 'octorus-16).
   ```

If you get a message like "Definition for SIMPLE does not exist," then you are probably in the wrong package. Execute `(pkg-goto 'cu t)` and try again.

4. When the `Observer` instrument window appears on the screen, a cursor will be blinking in the large pane in the middle. This is a Lisp Listener window, so you can type in any S-expression to be evaluated.

5. Middle button the mouse over the Lisp Listener to expose the simulator menu, and select the **Modify Simulation Parameters** choice. This wll pop up another menu corresponding to the parametric variations discussed in section 3.1.1. Make sure the 'production CARE run' choice is **Yes** and type in any comments you may have. Choose the **Exit** box in the margin after you are done.

6. To run the program, type `(obj-block)` to the Lisp Listener panel.

Chapter 4

Writing Applications to Run on CARE Machine Models

This chapter describes how to run example programs using the LAMINA programmer's interface. LAMINA itself is documented in chapter 2.

4.1 Examples to Run

This section provides further examples of how the LAMINA programming interface is used to write applications which run on CARE machine models. In particular, an object-oriented version of parallel Gaussian elimination is presented, which utilizes objects with sequential self-streams. Also, code is given for implementing some common shared variable synchronization mechanisms.

4.1.1 Object-Oriented Gaussian Elimination

As another example of how programs are written in the LAMINA object-oriented paradigm, this section presents a parallel implementation of Gaussian elimination with partial pivoting. The algorithm is described first, followed by portions of the serial and parallel implementations.

Column-Oriented Gaussian Elimination

This version of Gaussian elimination is *column-oriented*— a matrix is viewed as a collection of columns, each of which has an associated vector of data. To perform Gaussian elimination on an $N \times N$ matrix.

```
(defflavor SQUARE-MATRIX
          ((Columns)
           (Size 0)
           )
          ()
  :initable-instance-variables
  (:documentation "A square matrix manager."))

(defmethod (SQUARE-MATRIX :Init) (&rest ignore)
  "Create column vectors and the first column."
  (setf Columns (make-array Size))
  (loop for index from 0 below Size do
        (setf (aref Columns index)
              (make-instance 'Column :ID index
                                     :Size Size
                                     :Matrix Self)))
  (send Self :Initialize-Data))

(defmethod (SQUARE-MATRIX :Initialize-Data) ()
  (loop for index from 0 below Size
        as col-vector = (make-array Size :initial-element 0.0)
        do
        (send (aref Columns index) :Set-Data col-vector)))

(defmethod (SQUARE-MATRIX :Gauss) ()
  (let ((active-cols (listarray Columns)))
    (send (car active-cols) :Gauss-Pivot active-cols)))
```

Figure 4.1: Definition of Square-Matrix flavor.

loop for $i = 0$ to $N - 2$
 do pivot exchange for column i;
 forall $i \leq j < N$ and $i < k < N$,
 eliminate data in column j and row k;
 end forall;
end loop;

Sequential Object-Oriented Implementation

In the sequential implementation, two flavors are defined. Square-Matrix and Column. A Square-Matrix object contains a vector of Column objects, and a Column object contains a vector of data.

The definition of the Square-Matrix flavor is shown in figure 4.1. Upon creation, a Square-Matrix executes its :Init method, which creates its column objects and sends itself an :Initialize-Data message. The :Initialize-Data method initializes the data vector for each column to all zeroes. Since this isn't a very interesting matrix from a Gaussian elimination point of view, figure 4.2 shows the definition of the Test-Matrix flavor, which specializes the :Initialize-Data method to produce a lower diagonal matrix, initialized as shown below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots \\ 2 & 1 & 0 & 0 & \\ 3 & 2 & 1 & 0 & \\ 4 & 3 & 2 & 1 & \\ \vdots & & & & \ddots \end{bmatrix}$$

A Square-Matrix also responds to a :Gauss message. In this implementation, the matrix makes a list of all of its column objects and sends it in a :Gauss-Pivot message to the first column.

Each Column object has a ID instance variable, which indicates which column in the matrix it represents (starting with 0 for the left-most column). It also has instance variables which contain the data vector, the number of elements in the vector, and the owning matrix object. The definition of the Column flavor is shown in figure 4.3.

Column objects respond to two types of messages—:Gauss-Pivot and :Gauss-Step. These methods are show in figure 4.4.

A :Gauss-Pivot message tells a column to determine its *pivot*, which is the data element with largest absolute value, on or below the diagonal. The column then exchanges the diagonal element with the pivot and sends a :Gauss-Step message to all columns to its right. As arguments to the :Gauss-Step message, it sends its ID number, the row number of the pivot, and its column data. Finally, it sends a :Gauss-Pivot message to the column to its immediate right.

Upon receiving a :Gauss-Step message, the column performs the row exchange specified by the pivot. Then it eliminates its data elements on the rows below the pivot row.

```
(defflavor TEST-MATRIX
           ()
           (Square-Matrix)
  (:documentation "An invertible matrix."))

(defmethod (TEST-MATRIX :Initialize-Data) ()
  (loop for column from 0 below Size
        as col-vector = (make-array Size :initial-element 0.0) do
        (loop for index from column below Size
              for value from 1.0 do
              (setf (aref col-vector index) value))
        (send (aref Columns column) :Set-Data col-vector)))
```

Figure 4.2: Specialization of Square-Matrix.

```
(defflavor COLUMN
           (ID          ;;which column am I?
            Data        ;;column array
            Size        ;;column length
            Matrix      ;;owning matrix
            )
           ()
  :initable-instance-variables
  (:documentation "A column of a square matrix."))
```

Figure 4.3: Definition of Column flavor.

```
(defmethod (COLUMN :Gauss-Pivot) (active-cols)
  "Find pivot and send to other columns."
  (cond
    ((= (length active-cols) 1)
     (send Matrix :Display))
    (:else
     ;;== Find pivot. ==
     (let ((pivot ID)
           (new-diagonal (aref Data ID)))
       (loop for index from (1+ ID) below Size
             as element = (aref Data index)
             when (> (abs element) (abs new-diagonal))
             do (setq pivot index
                      new-diagonal element))
       ;;== Exchange Rows ==
       (unless (= pivot ID)
         (swapf (aref Data pivot) (aref Data ID)))
       ;;== Do elimination. ==
       (loop for obj in (cdr active-cols) do
             (send obj :Gauss-Step pivot ID Data))
       (array-initialize Data 0.0 (1+ ID) Size)
       (send (cadr active-cols) :Gauss-Pivot (cdr active-cols))))))

(defmethod (COLUMN :Gauss-Step) (pivot col-num pivot-column)
  "Receive pivot and update vector and eliminate local elements."
  (let ((pivot-value (aref pivot-column col-num))
        (pivot-row-value (aref Data pivot)))
    ;;== Exchange rows for pivot. ==
    (unless (= pivot col-num)
      (swapf (aref Data pivot) (aref Data col-num)))
    ;;== Eliminate elements below pivot. ==
    (loop for element from (1+ col-num) below Size
          for update-index from 1
          as value = (aref Data element)
          as col-value = (aref pivot-column element) do
          (unless (zerop col-value)
            (setf (aref Data element)
                  (- value
                     (* pivot-row-value
                        (// col-value pivot-value)))))))
```

Figure 4.4: Column flavor methods.

Parallel Implementation

Portions of the code for the parallel implementation of Gaussian elimination are shown in figures 4.5 and 4.6.

The first step toward converting the sequential implementation of Gaussian elimination described above to a parallel implementation is to redefine all the objects as LAMINA objects, by inheriting from the `lamina` flavor. This adds an extra instance variable, `Self-Stream`, which represents the task stream for the object. `Self-Stream` is initialized when a LAMINA object is created.

The next step is to determine how the objects should be created and initialized. As before, the `Square-Matrix` object does the creating—this time, however, data vectors are intialized *before* the column objects are created, to minimize the number of messages sent to the column objects. The `:Initialize-Data` method puts the data vectors in an instance variable called `Local-Data`; then the data vector is sent along with the creation message for a new `Column` object.

A `Square-Matrix` of size N will need to create N `Column` objects. When a `Column` is created, it sends a `:Reply` message, containing its `ID` and (a reference to) its `Self-Stream`, back to the matrix. For each `:Reply` message, the matrix updates its `Columns` instance variable. When it has received N replies, it posts its `Self-Stream` to its `Reply=>`[1] instance variable, indicating that it is completely initialized and ready for work.

In response to a `:Gauss` message, the `Square-Matrix` object collects a list of all the self-streams of the columns and sends it to all the `Column` objects. In this implementation, the columns each maintain a local copy of the list of active objects, to avoid the transmission cost of passing it with each pivot message. (In the sequential version, there was no significant cost with sending the list, since it was just a pointer to something on the heap. In the parallel version, there is no common heap—the list would be copied and transmitted every time.)

Another change in the parallel version is to recognize that only the data below the diagonal of the pivot column is needed for the eliminations in the other columns. (See the `:Gauss-Pivot` trigger in figure 4.6.) Also, whenever a column is finished pivoting, it sends its data to the matrix, which updates its local copy of the data.

Finally, to insure that each column performs its eliminations in sequential order, the `Column` flavor inherits from `sequenced-self-stream`, as well as from `lamina`. This means that the messages will be removed from the object's task stream in increasing order, according to their *tag* fields. Each `:Gauss-Pivot` and `:Gauss-Step` message is sent with its tag equal to the `ID` of the column doing the pivot.

The code for the `:Gauss-Pivot` method is shown in figure 4.6; the code for `:Gauss-Step` is essentially the same as for the sequential version, shown in figure 4.4.

[1] The symbol "=>" is a mnemonic for "stream," so `Reply=>` should be read "reply-stream."

```
(defflavor SQUARE-MATRIX
           ((Columns)    ;;array of column self-streams
            (Size 0)     ;;number of columns
            (Local-Data) ;;local copy of column data
            (Processors) ;;number of processors to use
            (Replies 0)  ;;tally of column creation replies
            (Reply=>))   ;;stream to reply to after creation
           (lamina)
  :initable-instance-variables
  (:documentation "A square matrix."))

(defmethod (SQUARE-MATRIX :After :Init) (&rest ignore)
  "Create column vectors and initialize data."
  (setf Columns (make-array Size))
  (setf Local-Data (make-array Size))
  (send Self :Initialize-Data)
  (loop for index from 0 below Size
        as site = (aref c:***All-Sites-Vector***
                        (mod index Processors)) do
        (creating 'Column '(:ID ,index :Size ,Size
                            :Matrix ,Self-Stream
                            :Data ,(aref Local-Data index))
                  on site)))

(deftrigger (SQUARE-MATRIX :Reply) ((column-obj column-id))
  "Enter object into Columns array."
  (setf (aref Columns column-id) column-obj)
  (when (= (incf Replies) Size)
    (posting Self-Stream to Reply=>)))

(deftrigger (SQUARE-MATRIX :Gauss) ()
  (let ((active-cols (listarray Columns)))
    (sending active-cols :Set-ActiveCols active-cols by -1)
    (sending (car active-cols) :Gauss-Pivot nil by 0)))
```

Figure 4.5: Square-Matrix code for parallel Gaussian elimination.

```
(defflavor COLUMN
        (ID          ;;which column am I?
        Data         ;;column array
        Size         ;;column length
        Matrix       ;;owning matrix
        (ActiveCols)
        )
        (sequenced-self-stream lamina)
 :initable-instance-variables
 (:documentation "Column of a square matrix."))

(defmethod (COLUMN :After :Init) (&rest ignore)
 (sending Matrix :Reply '(,Self-Stream ,ID)))

(deftrigger (COLUMN :Gauss-Pivot) ()
  "Find pivot and send to other eliminators."
  (pop ActiveCols)
  ;;== Find pivot. ==
  (let ((pivot ID) (col-vector (make-array (- Size ID)))
        (new-diagonal (aref Data ID)))
    (loop for index from (1+ ID) below Size
          as element = (aref Data index)
          when (> (abs element) (abs new-diagonal))
          do (setq pivot index
                   new-diagonal element))
    ;;== Exchange Rows ==
    (unless (= pivot ID)
      (swapf (aref Data pivot) (aref Data ID)))
    ;;== Do elimination. ==
    (copy-array-portion Data ID Size col-vector 0 (- Size ID))
    (when ActiveCols
      (sending ActiveCols :Gauss-Step
               '(,pivot ,ID ,col-vector) by ID))
    (array-initialize Data 0.0 (1+ ID) Size)
    (sending Matrix :Update-Local-Data '(,ID ,Data))
    (when ActiveCols
      (sending (car ActiveCols) :Gauss-Pivot nil by (1+ ID)))))
```

Figure 4.6: Column code for parallel Gaussian elimination

4.1.2 Shared Variable Synchronization Mechanisms

The shared variable ordering example, presented in chapter 2, used two types of synchronization mechanisms—spin locks and shared queues. In this section, we show how other synchronization mechanisms may be implemented in terms of shared variables and queues.

Distributed Spin Lock

In LAMINA, a simple (binary) spin lock may be implemented by initializing a shared variable to some non-NIL value. To acquire the lock, a process exchanges the value NIL with the current contents of the variable—the lock is acquired when the value returned is non-NIL. To release the lock, the process writes a non-NIL value into the variable. This implementation is shown in the definition of `with-spin-lock`, shown in figure 4.7.

```
(defmacro WITH-SPIN-LOCK (lock &body body)
  (let ((lock-name (gensym)) (value-name (gensym)))
    `(let ((,lock-name ,lock))
       (loop until (shared-exchange ,lock-name nil)) ;acquire
       (let ((,value-name (progn ,@body)))
  (shared-write ,lock-name t) ;release
  ,value-name))))
```

Figure 4.7: Simple spin lock.

Spinning on a single location can cause severe contention at that variable's memory module. In the CARE reference passing machine model, if many processors are requesting the same variable, the requests may build up in the fifo-buffer feeding the memory controller (the operator). This means long latencies for the requests, and it also means that the write which will release the lock must wait for all the reads ahead of it to complete. Another problem is fairness—the processors which are physically closer to the memory module containing the lock will have more chances to acquire the lock than distant processors.

To alleviate these problems, we define a *distributed spin lock*. A distributed spin lock for P processors involves $2P + 1$ shared variables, distributed throughout the memory modules in the system. One shared variable, called the *key*, contains the identifier of the last processor to request the lock. The other variables, called *sub-locks*, are locations which grant the lock. Each processor requesting the lock will spin on a different sub-lock. Also, each processor keeps a local array which contains the references to all the sub-locks.

For the moment, assume that each processor has a unique *id*, where $0 \leq id < P$. The operation of the spin lock is as follows:

1. To *acquire* the lock:
 (a) Exchange *id* with value of *key*; store the result in *last-key*.

(b) If *last-key* = NIL, then no processors has ever requested the lock, so it is acquired.

(c) If *last-key* ≠ NIL, then exchange NIL with the value of the *sub-lock* associated with *last-key*. When a non-NIL value is returned, the lock is acquired.

2. To release the lock, write T (or some non-NIL value) to the sub-lock associated with *id*.

Unfortunately, the above procedure may result in an undesirable race condition, as shown in figure 4.8. After step (6), processors 1 and 2 are both spinning on sub-lock 0. If processor 2 reads the lock first, as shown in step (7), processors 0 and 1 will spin forever. To eliminate this problem, we must provide $2P$ sub-locks, rather than P, and each processor must alternate between setting *id* and $id + P$ into the *key* location. In terms of our example, processor 2 would spin on sub-lock 4, so the race is avoided.

Figure 4.9 shows an implementation of a distributed spin lock. The function `distributed-spin-lock` returns a closure which may be used to acquire and release the lock.[2] The macro `with-distributed-spin-lock` is used to delineate critical regions, similar to `with-spin-lock`.

[2] As implemented, the caller of the closure must supply an *id* and keep track of whether this is an "even" or "odd" access. Alternatively, each processor could create its own local closure which handles the bookkeeping.

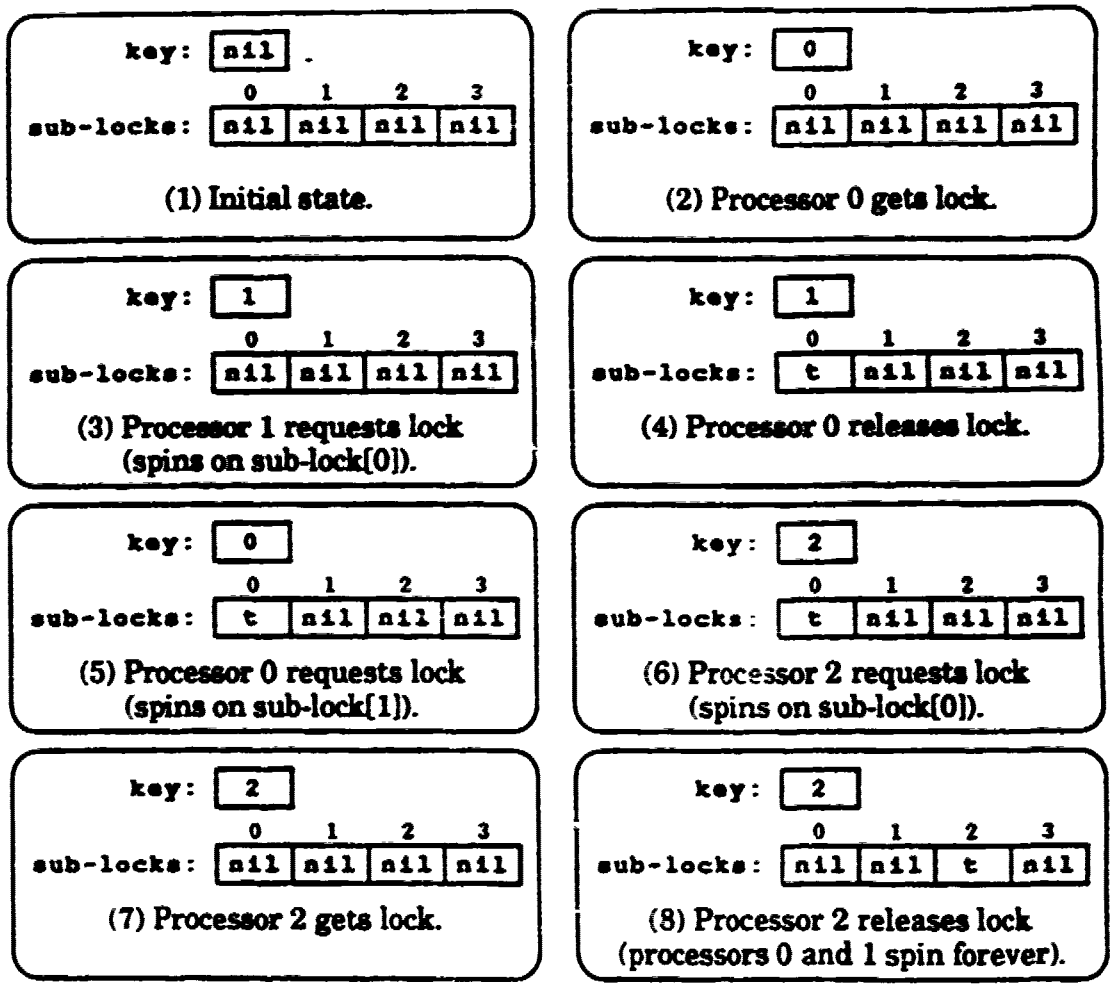


Figure 4.8 Race condition in a naive distributed spin lock.

```
(defun distributed-spin-lock (&rest ignore)
  (let ((num-locks (length ***All-Processors***)))
    (let ((<key> (shared-variable nil (random-memory)))
          (lock-array (make-array (* 2 num-locks))))
      (loop for site in ***All-Processors***
            as index = (get-local-thread-number site t)
            as memory = (send site :get-associated-memory) do
            (setf (aref lock-array index) (shared-variable nil memory))
            (setf (aref lock-array (+ index num-locks))
                  (shared-variable nil memory)))
      #'(lambda (operation id-num alternate?)
          (selectq operation
            (:acquire
              (let ((last-owner
                      (shared-exchange
                        <key> (+ id-num (if alternate? num-locks 0)))))
                (when last-owner
                    (loop until (shared-exchange
                                  (aref lock-array last-owner)
                                  nil)))))
            (:release
              (shared-write
                (aref lock-array (+ id-num (if alternate? num-locks 0)))
                t)))))))

(defmacro WITH-DISTRIBUTED-SPIN-LOCK (lock id flag &body body)
  (let ((lock-name (gensym)) (value-name (gensym)))
    `(let ((,lock-name ,lock))
       (funcall ,lock-name :acquire ,id ,flag)
       (let ((,value-name (progn ,@body)))
         (funcall ,lock-name :release ,id ,flag)
         ,value-name))))
```

Figure 4.9: Implementation of distributed spin lock.

```
(defmacro SEMAPHORE (&optional (initial-value 1))
  `(let ((semaphore (shared-queue '(:semaphore :init ,initial-value))))
     (loop repeat ,initial-value do
           (shared-enqueue semaphore semaphore))
     semaphore))

(defun P (semaphore) (shared-dequeue semaphore))
(defun V (semaphore) (shared-enqueue semaphore semaphore))

(defun MUTEX-SEMAPHORE () (semaphore 1))
(defun WAIT-SEMAPHORE () (semaphore 0))
(defun COUNT-SEMAPHORE (num) (semaphore num))
```

Figure 4.10: Implementation of semaphores.

Shared-Queue Semaphores

In the case where more than one process is running on a processor, it may be more efficient to deschedule a process while waiting on a lock and reschedule it when the lock is acquired. Furthermore, the programmer may want to deal with higher level constructs than spin locks. In this section, we consider how to implement *semaphores* using shared queues.

One view of a semaphore is as a non-negative integer-valued variable. Two operations are defined on semaphores. `P` waits until the value is greater than zero, then decrements it[3], `V` increments the value by one. Depending on how its value is initialized, a semaphore may be used for *waiting* (intialize to 0), *mutual exclusion* (initialize to 1), or *counting* (initialize to n, the number of "events" to be counted).

Because the value of the semaphore is not directly accessible by the process operating on it, it does not strictly have to be implemented in terms of incrementing or decrementing an integer. In particular, we will implement a semaphore in terms of a shared queue of *tokens*, where the number of tokens represents the "value." (In this implementation, the token is a reference to the semaphore itself.)

Figure 4.10 shows the implementation of a general semaphore in terms of shared queues. When a process executes a `P` operation, it sends a `shared-dequeue` request (and gets descheduled). If there is a token on the queue, it will be removed from the queue and sent to the requesting process, which will then be rescheduled. If there is no token, the process will be added to a queue of requestors—when a token is placed on the queue (by a `V` operation), the request at the head of the request queue is serviced. Thus, access to the semaphore is guaranteed to be fair, since the requests are handled in the order in which they arrive.

[3]The test and decrement must be atomic.

Barriers

Another common synchronization mechanism in shared-variable programming is the *barrier*, which represents a synchronization point among several processes. When a process reaches a barrier, it waits until *all* the processes have reached it. There may then be some critical section of code which is executed *only* by the last process to arrive at the barrier. After the critical section is executed, all the processes may proceed.

Figure 4.11 shows how barriers may be implemented out of a counter and two spin locks. The function `barrier` returns a shared-array of four elements, three of which are references to shared variables. The first element (`<entry-lock>`) is used as a spin lock to gain access to the barrier counter. The second element (`<count>`) is used to count the number of processes which have not yet reached the barrier—this variable is initialized by the call to `barrier` and is reset when the critical region is exited. The third element (`<exit-lock>`) is used as a spin lock to determine when it is safe to continue execution. The fourth and final element is an integer which represents the initial value of the counter.

The `with-barrier` macro specifies both the barrier and the code in the critical region. If the barrier passed to the macro is a shared array, it is "cached" for fast access to the locks and counter. Alternatively, the process can cache the shared array once and pass a local array (containing references) to the `with-barrier` form to avoid repeated caching.

Within the `with-barrier`, the process first gets exclusive access to the counter and decrements it, if it is not yet zero. Also, if this is the first process to reach the barrier, it sets `<exit-lock>` to `NIL`. Finally, `<entry-lock>` is released, and the process waits for `<exit-lock>` to have a non-`NIL` value.

If the counter was zero, then this is the last process to reach the barrier. The critical code is executed, the counter is reset, and the `<exit-lock>` is released. At this point, all the processes may proceed.

```
(defun BARRIER (num-processes)
  (let ((local-barrier (make-array 4)))
    (setf (aref local-barrier 3) (1- num-processes))
    (setf (aref local-barrier 0) (shared-variable
                                    (1- num-processes)))
    (setf (aref  ocal-barrier 1) (shared-variable t))
    (setf (ar·   .ocal-barrier 2) (shared-variable nil))
    (shared-array 4 :initial-contents local-barrier)))

(defmacro WITH-BARRIER (barrier &rest body)
  (let ((<count> (gensym)) (<entry-lock> (gensym))
        (<exit-lock> (gensym)) (init-value (gensym))
        (value-name (gensym)) (local-barrier (gensym))
        (count-value (gensym)) (temp-count (gensym)))
    '(let* ((,local-barrier
             (if (c:remote-address-p ,barrier)
                 (cache-shared-array ,barrier)
                 ,barrier))
            (,<count> (aref ,local-barrier 0))
            (,<entry-lock> (aref ,local-barrier 1))
            (,<exit-lock> (aref ,local-barrier 2))
            (,init-value (aref ,local-barrier 3)))
       (let ((,count-value
              (with-spin-lock ,<entry-lock>
                (let ((,temp-count (shared-read ,<count>)))
                  (when (= ,temp-count ,init-value)
                    (shared-write ,<exit-lock> nil))
                  (when (> ,temp-count 0)
                    (shared-write ,<count> (1- ,temp-count)))
                  ,temp-count))))
         (cond
           ((> ,count-value 0)
            (loop until (shared-read ,<exit-lock>)))
           (:else
            (let ((,value-name (progn ,@body)))
              (shared-write ,<count> ,init-value)
              (shared-write ,<exit-lock> t)
              ,value-name)))))))
```

Figure 4.11: Implementation of barrier synchronization.

Chapter 5

CARE System Design

This chapter describes how to build a *design*[1]—that is, a system of components to be simulated—out of the CARE components supplied with the system.

The basic tool provided for describing the structure of a design is the P-HELIOS structure editor. P-HELIOS allows you to create a design by using components from a library of prototypes and connecting them in any arbitrary topology. For the purposes of this chapter, the library of components to be used is provided by CARE. (A later chapter will describe how to design and build your own component library.) Section 5.1 gives a brief tutorial on how to use P-HELIOS—a more complete description of the commands available can be found in the 'Helios User Manual' supplied as an appendix.

The basic building block of CARE designs is the *site* component. A site represents a processor-memory pair together with communications hardware. (The subcomponents of a site are described in 'A Dynamic Cut-through Communications Protocol with Multicast', supplied as an appendix.) Building a multiprocessor design involves connecting sites together with communications channels in some topology.

Sites may be connected using *manual*, *semi-automatic*, or *automatic* wiring facilities. Automatic wiring is currently supported for grid, torus, bus, and two-level hierarchical bus topologies. These represent all the pre-defined multiprocessor organizations currently provided by CARE. The tools provided for automatic wiring are described in section 5.2, along with several examples of how to create new designs.

At the other extreme, P-HELIOS allows you to connect components together manually, by drawing lines between communication ports. This facility, described in section 5.3, would be used by a user who wants to connect sites in some topology which is not currently supplied by CARE, or by those users who wish to create their own component libraries.

Finally, you may automatically iterate any component in a two-dimensional grid pattern, by choosing the component and manually specifying the connections between neighbors. Both the component and the specified connections will be iterated when the design is loaded. This approach is called semi-automatic wiring and is described in section 5.4.

[1]The term *design* was chosen to avoid confusion with the Lisp machine concept of a (software) *system*.

5.1 The P-HELIOS Structure Editor

The P-HELIOS structure editor allows you to graphically specify the structure of a design to be simulated. Designs are built by connecting together predefined components (*prototypes*) from a component *library*. Components are represented as boxes. Boxes may have *ports*, which may be connected by *lines* (representing communications channels) to the ports of other boxes. Also, boxes may contain other boxes, representing sub-components.

Before building a design, you must load a component library. This library specifies the prototype components which may be used to build designs. You may add new prototypes to the library, or edit existing ones. In addition, a design may be "prototized," added to the library, and used as a component in a later design. A design or library may be saved to a file for later use.

The remainder of this section explains how to get started with P-HELIOS and provides a short tutorial about how to use the editor.

5.1.1 Getting Started

The following steps are necessary to use the P-HELIOS editor:

1. **Load the P-HELIOS system.**

 Executing (`make-system 'p-helios :noconfirm :silent :nowarn`) loads all the files necessary to run the editor.

2. **Load component definitions.**

 P-HELIOS needs to know about the definitions of all the library components before it can actually load the library. If the CARE system is already loaded, then nothing else needs to be done for this step. Otherwise, execute the following:

   ```
   (make-system 'care-components :noconfirm :silent :nowarn).
   ```

3. **Invoke the editor.**

 This may be done by typing SYSTEM-h, or by executing (`s:p-helios`). Two windows will be displayed (see figure 5.1): a small one, called the *interaction window* for status reports and queries by the editor, and a large one, called the *main screen*, used for editing.

4. **Load the library.**

 If you are going to build designs using P-HELIOS, you need to load a component library. Clicking left over the main screen brings up the *Editor Operations* menu, shown in figure 5.2. Click over the *Load Library* command.

 At this point, the message `Enter library name:` appears in the interation window. Type `care-all`[2], followed by a carriage return. A verification menu will then appear, showing the default pathname for

[2]The library named `care-all` contains all the components needed to create the designs described in this chapter. There are other libraries which contain *only* the components needed to build certain types of designs — for example, the `care-bus` library contains only the components needed to build bus designs. All the predefined libraries can be found in the `care libraries;` directory.

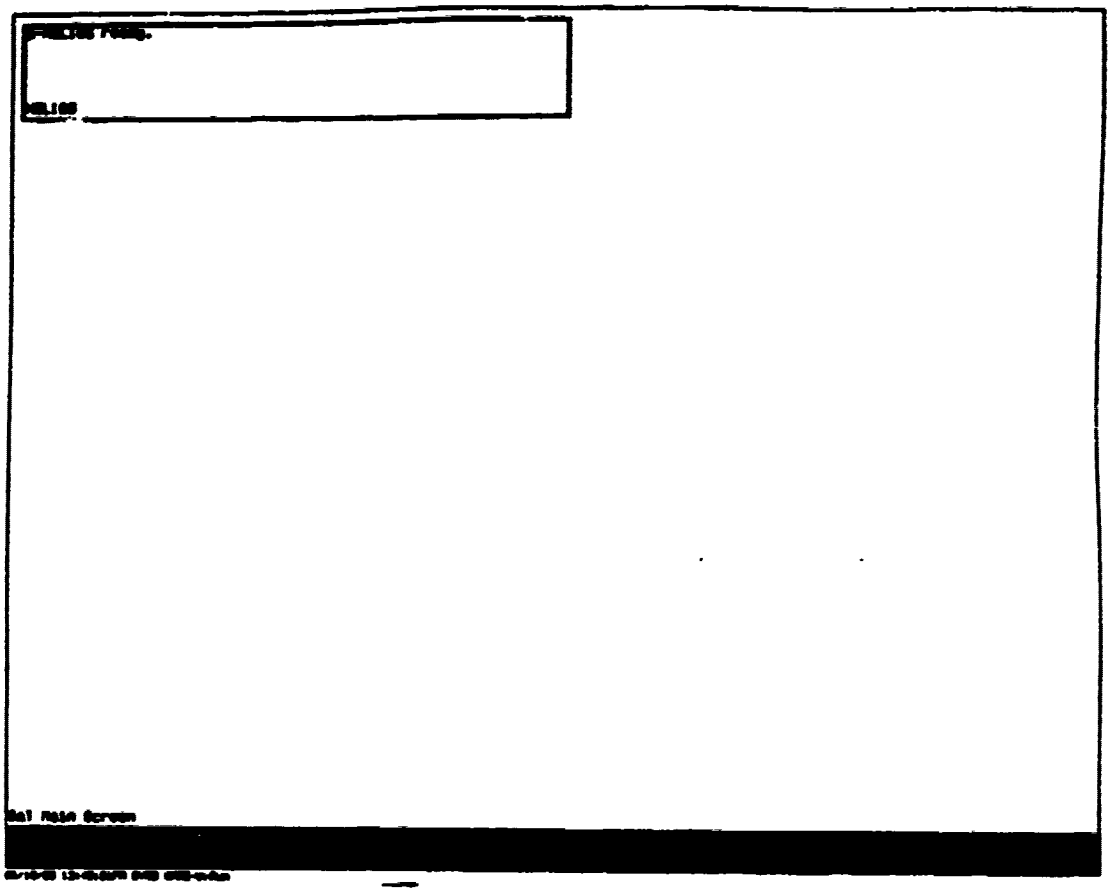


Figure 5.1: Initial P-HELIOS screen.

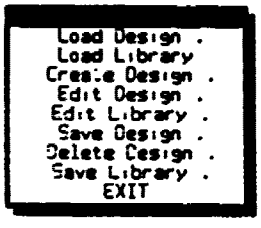


Figure 5.2: P-HELIOS editor operations

the library you have named. Select *No*. A message will appear in the interaction window, asking for the correct pathname. Type

```
care:libraries;care-all.x.
```
[3]

The verification menu will appear again, with (hopefully) the correct pathname, so you should select *Yes*, and the appropriate library file will be loaded.

5.1.2 Using P-HELIOS: A Brief Tutorial

After the steps in the previous section have been completed, you are ready to create a new design, or to edit an existing one. The following tutorial will lead you through the steps of viewing and manipulating designs. Following sections will describe the wiring facilities which allow new designs to be created.

Loading a Design

To edit a design that was previously saved in a file, you must first *load* the design into the editor, as follows:

1. Click left over the main screen to get the *Editor Operations* menu (figure 5.2), and select the *Load Design* command. This is very similar to loading a library, as in the previous section.

2. The interaction window will prompt you to provide the name of a design. Type `octorus-9` This will load a nine-element torus, in which each site is connected to its eight neighbors.

3. A verification window will appear, showing the default pathname for the design. As this will almost certainly be wrong, selec *No*. In response to the interaction window prompt, type `care:designs;octorus-9.x`.[4] When the verification window appears again, check the displayed pathname, and select *Yes* if it is correct.

4. When the file is loaded, a message like

   ```
   Design OCTORUS-9 loaded.
   ```

 will be displayed in the interaction window. Loading a design file sometimes takes a long time, especially if the design contains a lot of components. (Loading `octorus-9` should take a minute or so.)

Editing a Design

To edit a design which has previously been loaded (or created), you must first create a window in which to view the design.

[3] As a convention, design files and library files have extension `.x`, rather than `.lisp`.

[4] You need only type the portion of the pathname which differs from the default. For example, if the default were `care:designs;octorus-4.x`, you could type `octorus-9` to get the desired file.

1. Click left on the main screen and select the *Edit Design* command from the *Editor Operations* menu. A menu showing all the defined boxes[5]—select *Octorus-9*.

2. The mouse cursor will change to "+," and the following message will appear in the interaction window:

   ```
   Define the screen region for OCTORUS-9's viewport.
   ```

 This means that you should define the borders of the window which will be used for viewing and editing the `octorus-9` design, as explained in the next two steps.

3. Move the mouse cursor to the position on the main screen where you want to place the upper, left-hand corner of the viewing window. Then click left. Similarly, select the position of the lower, right-hand corner and click left. A dotted-line box outlines the shape and position of the screen as you've specified it so far.

 If you're not happy with the position of the lower, right-hand corner, you can reposition it by clicking (or dragging) the left button. If you want to reposition the upper, left-hand corner, first click the middle button, and then click (or drag) the left button. Clicking the middle button always allows you to reposition the other corner.

4. When you are satisfied with the size and placement of the viewing window, click right to confirm it.

5. At this point, if you set the display level to 2 (see the next section), the screen should look similar to figure 5.3. The name of the design (`octorus-9`) is displayed at the bottom, left-hand corner of the window, as well as the number of *levels* of the composition hierarchy currently being displayed.

Viewing the Design

Clicking the right button over the design's viewport window will bring up a menu of *Window Operations*, shown in figure 5.4. These allow you to change how the design is displayed on the screen. A few of the most important operations are discussed below.

- *Set Display Level:* This command determines which levels of the design's composition hierarchy will be drawn on the screen. At *level* = 1, the `octorus-9` design is displayed only as two boxes (one for the outer-level box, and one for the box that contains the sites—see section 5.2).

 Selecting *Set Display Level* from the *Window Operations* menu results in a menu of numbers, from which you can select the desired display level. Notice how the display for the `octorus-9` design changes when *level* = 2 and *level* = 3.

- *Zoom In:* This command will expand a portion of the current display to fill the entire viewing window. A menu is presented which allows you to select an expansion factor (e.g., selecting 2 makes everything displayed as twice the current size).

 The mouse cursor then changes to a box which represents the portion of the current display which will be visible after "zooming in." Move the mouse to position the box to include the portion of the display that you want to see, then click left.

[5] *Box* is a P-HELIOS synonym for *design*.

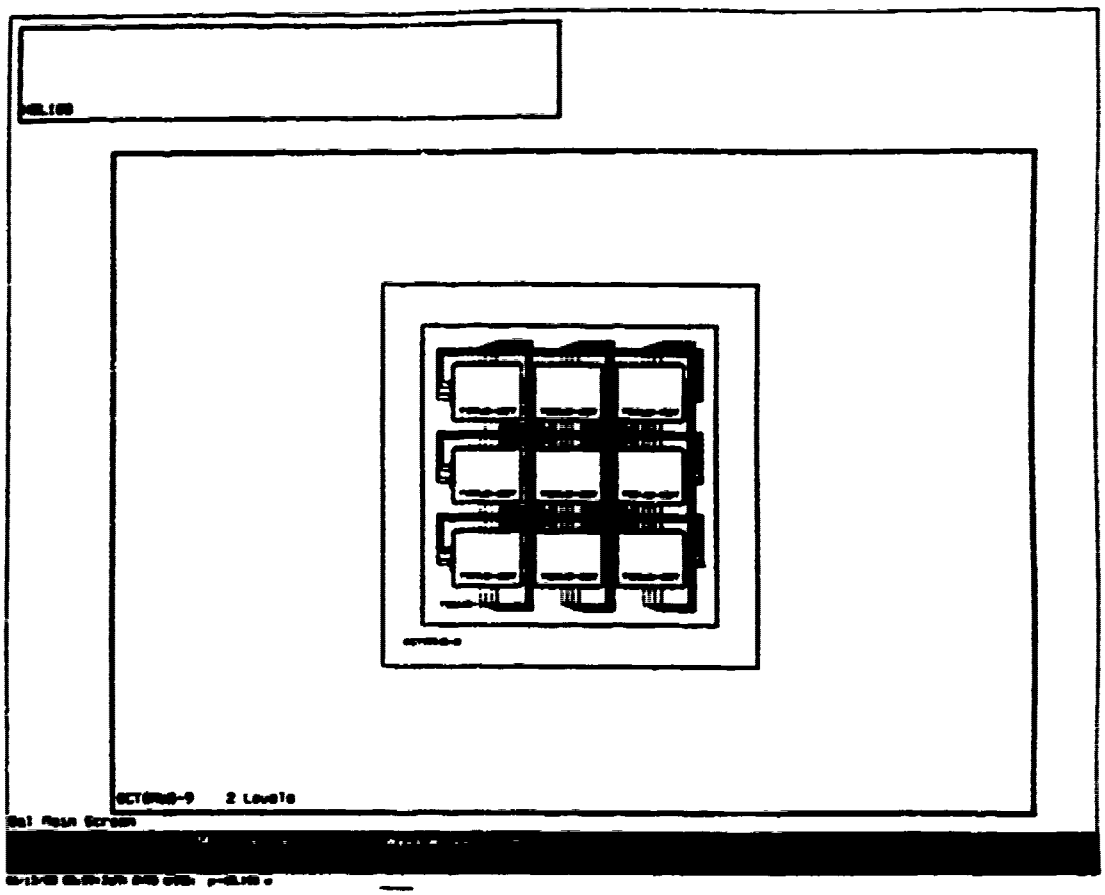


Figure 5.3: Editing a CARE design.

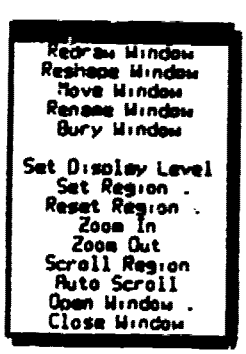


Figure 5.4: Design viewing commands

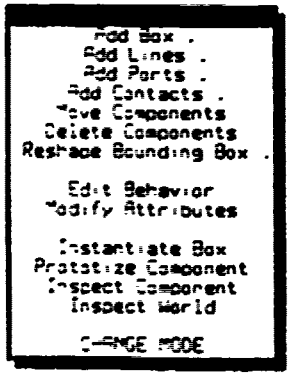


Figure 5.5 Design editing commands

- *Zoom Out:* This is the opposite of *Zoom In*—the design is displayed at a fraction of the current size, so more of the design will fit within the viewing window. Again, a menu is presented for you to select a zooming factor.

- *Scroll Region:* This command allows you to change the relative position of the region being displayed. The mouse cursor becomes "x"—click left on a position that you want to remain in the viewing window.

 Move the mouse to change the display location of the position you selected, and click right to confirm. The design will be redrawn in its new position.

Changing the Design[6]

Clicking left over the design's viewing window brings up a menu of *Edit Operations*, shown in figure 5.5. This section will describe some of the commands used to create CARE designs.

- *Delete Components:* This command removes a component (and its subcomponents) from the current design. The component may be a box, a line, or a port. Only an entity which is currently visible may be deleted.

 1. Select the *Delete Components* command. The mouse cursor changes into a small x, and the following message is printed in the interaction window:

 `Choose a subcomponent to delete (Middle button if done deleting).`

 2. Place the mouse cursor over one of the sites in **`octorus-9`** and click left. The box should become highlighted, which means that this box has been chosen for deletion.
 3. Click right to confirm the choice (or middle to abort), and the site will disappear, along with any wires connected to it.
 4. When you are finished deleting components, click the middle button to quit.

- *Add Box:* This command is used to add a component to the design being edited. Selecting this command with the left or middle mouse button adds the box to the highest level box in the design (e.g., the box labelled **`octorus-9`**). If the right button is used instead, you must select a box to own the new component.

 Two steps are involved in adding a new component: (1) selecting the prototype of the component to add, and (2) placing the new component in the existing design.

 1. Click left on the *Add Box* command.
 2. An menu will appear which lists the prototypes defined by the currently loaded library. Select the type of component you wish to add, e.g., *torus-site*.
 3. The system will then create a *box-descriptor* corresponding to the selected prototype. A menu will appear which will ask if you want to use the default name for the box- usually something like **`torus-site-1`**. Selecting *No* allows you to name the component yourself

[6]Sometimes, while editing a design, the mouse cursor will become "trapped" in the viewing window—i.e., you will not be able to move the mouse cursor outside the window. When this happens, you should save PHELIOS and then return, e.g., by SYSTEM-L SYSTEM-E. When you return to PHELIOS, things should be back to normal.

4. Place the new component by holding down the left button and "dragging" the box. Clicking the middle button allows you to scale, rotate, or flip the object.
5. Click right to confirm the component's location.

- *Moving Components:* This command is for changing the position of a currently visible component (box, port, line, ...).
 1. First, select the component to be moved by clicking left on it, as in the *Delete Components* command. Click right to confirm the selection.
 2. Move the component by holding down the left button and dragging it. The middle button provides scaling, rotations and reflections, as in the *Add Box* command. Confirm the new position by clicking right.
 3. When done moving components, click the middle button.
- *Reshape Bounding Box:* This command allows the outer level box to be refined, in the same way that you defined the shape and position of the viewing window above.
 1. Click left to fix the upper, left-hand corner.
 2. Click left to fix the lower, right-hand corner.
 3. Click right to confirm the new bounding box.

Some of the other editing commands will be described in the wiring sections below.

Creating a New Design

To create a new design to be edited, select the *Create Design* command from the *Editor Operations* menu (figure 5.2)

A menu will appear asking if you want to use the default name for the new design, which is usually something like `Box-1`. Select *No* to specify your own name.

Then you will be asked to define a viewing window for the new design, as in the previous section on *Edit Design*. When the viewing window is defined, a single, empty box will be displayed - this is the outer-level box for the new design. Use the editing commands described above to reshape the box, add subcomponents, and so forth.

Saving a Design

When you've finished editing a design, you may save it to a file.[7] Select the *Save Design* command from the *Editor Operations* menu (figure 5.2). A menu will appear, listing the currently defined designs - select the one that you wish to save.

[7] Note that currently the only way to simulate a design is to first save it to a file and then load that file using `Simple`.

A verification window will appear, asking you to confirm the default name of the output file. Select *No* to specify your own file. Remember that design files, by convention, should use the `.x` extension, rather than `.lisp`.

5.2 Automatic Wiring

Wiring is the term used to describe connecting components together in a design by drawing lines between their communications ports. For some extremely regular designs, mechanisms for *automatic wiring* have been developed, in which all placement and connections are performed by the system. The topologies supported by automatic wiring in CARE are grid, torus, single-level bus, and two-level bus.

In addition to sparing you the onerous task of wiring up large, repetive designs, the automatic wiring facilities also save on file space. Instead of saving fully instantiated designs, which can be quite big, you save a *generator* component. When this component is ***instantiated***, it creates instances of all the desired components and names, places, and connects them appropriately. This instantiation usually happens when the design file is *loaded*, but you can also instantiate designs within P-HELIOS, to view or edit the fully created structure.

To use automatic wiring, you simply add an appropriate component, such as a *Bus-Box*, and specify the parameters of the system to be built. All the subcomponents required for the system are automatically copied from the library—when the component is instantiated, the subcomponents are replicated and connected by P-HELIOS.

The following examples show how to used the automatic wiring facilities to create grid, torus, and bus designs. In addition, design files themselves may be used to create new designs, bypassing the editor altogether. This approach is described in section 5.2.4.

5.2.1 Example: Building a Grid or Torus

The simplest topology supported by CARE is the *grid*—a collection of *site* components, wired in a two-dimensional array. A *torus* is a grid whose connections are "wrapped around" the edges of the array. Each processing element in a torus design is actually *torus-site*—a specialization of a site whose routing methods are changed to include the wraparound connections.

To create a CARE torus design, create a new design, as in the previous section, and do the following:

1. Invoke the *Add Box* command (from the *Edit Operations* menu—figure 5.5) and select the *Torus*[8] protoype from the library.
2. A menu will appear, asking you to select the *Grid Type* for the design. You may select *Quad*, *Hex* or *Octal*, depending on whether each site should be connected to four, six, or eight neighbors, respectively. (In a CARE torus, diagonal connections are not "wrapped around.")

[8] To build a grid (no wraparound connections on the edges), select the *Grid* prototype instead of *Torus*.

3. Next, another menu will appear, listing the other attributes of the torus which may be specified.

 (a) Select the *Dimensions* entry from the attributes menu. This allows you to specify the number of rows and columns of the torus. For example, if you want to build a 4 × 4 torus, you should respond to the prompts in the interaction window as follows:

   ```
   Number of columns:  4
   Number of rows:  4
   ```

 (b) If you are building this design to be saved to a file for simulation at a later time, select the *Instantiate When Loaded* entry from the attributes menu. Answer tt y to the prompt in the interaction window. This specifies that design should be instantiated when the file is loaded (see above).

 (c) Move the mouse away from the attributes menu, and it will disappear.

4. Finally, a box will appear, and you will be asked to place it in the upper, left-hand corner. Move the mouse to place the component, then click right to confirm.

5. If the design is to be saved now is the time to do so. If you want to view (or edit) the design in its fully instantiated state, proceed to the following step. (This can also be done *after* the design has been saved.)

6. Select the *Instantiate Box* command from the *Edit Operations* menu, and select the torus box. P-HELIOS will go busy for while, creating and wiring the new design. When "Done!" appears in the interaction window, the design has been instantiated. Use the commands in the *Window Operations* menu (figure 5.4) to view the design—you will probably have to zoom out to see the whole thing.

5.2.2 Example: Building a Bus

This section describes the steps required to build a single-level bus design The components which are connected to the bus are specializations of the site components, called *bus-sites*. Bus-sites have no network interface components, except for fifo-buffers, because there is no routing required within the site to get to the bus (since it is connected to only one bus). The bus component uses net-inputs and net-outputs (described in the appendix 'A Dynamic Cut-through Communications Protocol with Multicast' supplied as an appendix) to implement the CARE routing protocol—see figure 5.6.

To build a CARE bus, create a design and do the following:

1. Select the *Add Box* command from the *Edit Operations* menu (figure 5.5), and select the *Bus-Box* prototype from the library. Use the default name (or whatever you like).

2. Next, a menu will appear which allows you to specify the parameters of the bus

 (a) Select *Bus Structure* from the attributes menu—another menu will appear, giving you the option of creating a single-level or two-level bus. Select a single-level bus, and type in the number of sites in response to the prompt in the interaction window.

 (b) If you are planning to save this design to a file, select *Instantiate When Loaded* from the attributes menu, and type y in response to the prompt in the interaction window. This specifies that the design should be instantiated when the file is loaded

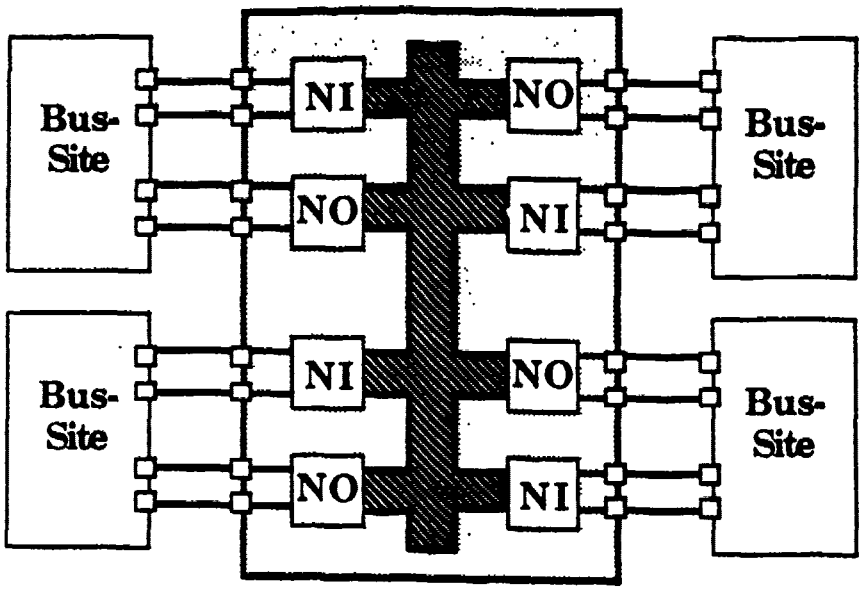


Figure 5.6: CARE b: :ponent.

(c) Move the mouse cursor away from the attributes and will disappear.

3. If the design is to be saved. now is the time to do so. If you wa: 'o view (or edit) the design in its fully instantiated state, proceed to the following step. (This can a:- be done *after* the design has been saved.)

4. Select the *Instantiate Box* command from the *Edit Operations* menu, and select the bus-box. P-HELIOS will go busy for while, creating and wiring the new design. When "Done!" appears in the interaction window, the design has been instantiated. Use the commands in the *Window Operations* menu (figure 5.4) to view the design—you will probably have to zoom out to see the whole thing

5.2.3 Example: Building a Two-Level Bus

In addition to a single, global bus. CARE supports a two-level hierarchy of busses *Clusters* of sites, each with its own local bus, are connected to a global bus for intercluster communication.

Because of the combination of unbounded packet size and cut-through routing supported by the CARE communications protocol, deadlock can occur if simple busses are used. Consider the case. shown in figure 5.7, where a site in one cluster (*cluster-1*) is trying to communicate with a site in another cluster (*cluster-2*), and, at the same time. a site in *cluster-2* is trying to communicate with a site in *cluster-1*. Each site grabs its local bus and then tries to access the global bus. Assume that the site from *cluster-1* gets the global bus first—it will not be able to continue. since the bus in *cluster-2* is busy. Furthermore, the *cluster-2* bus will not go free until it gets access to the global bus. Thus. there is a cycle in the resources required to complete the communication, resulting in deadlock.

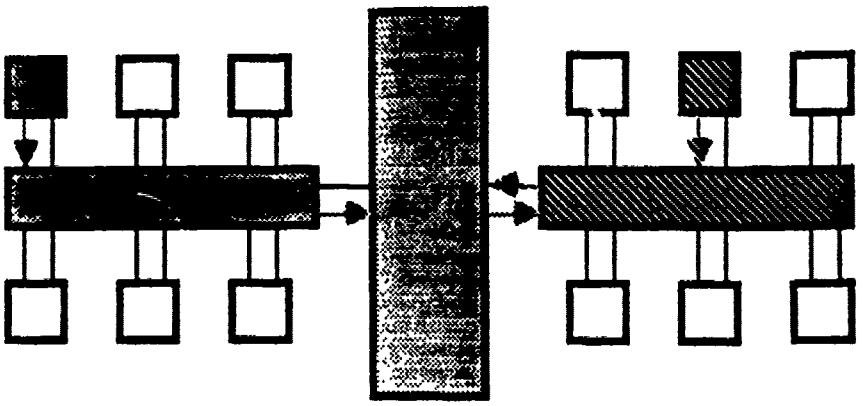


Figure 5.7 Deadlock in a two-level bus

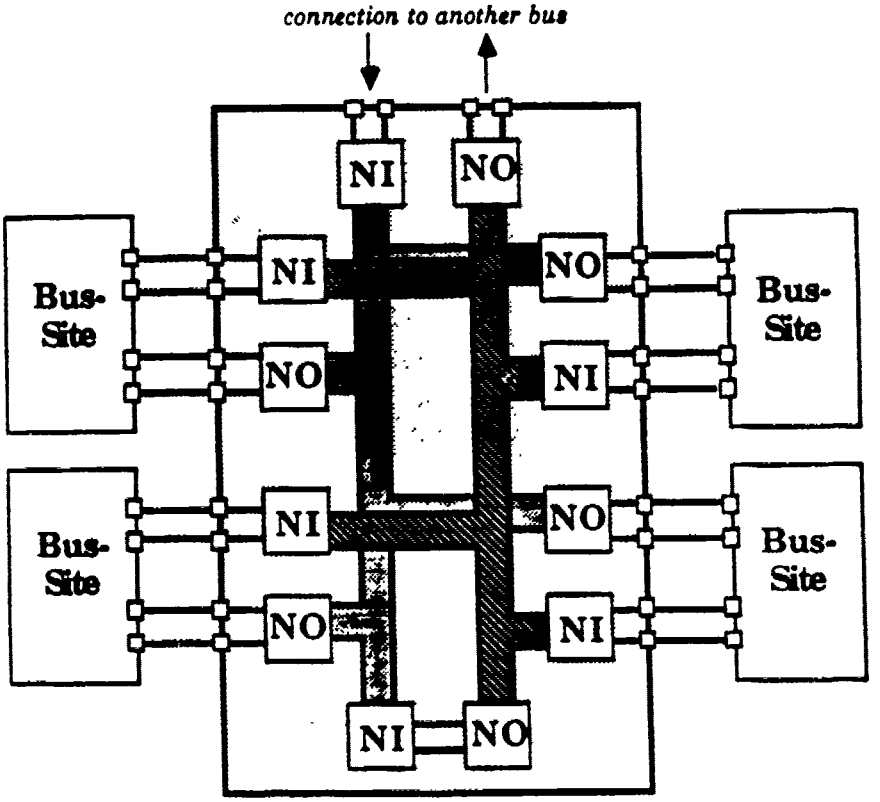


Figure 5.8 CARE dual-bus component

To avoid this possibility, each local bus is implemented as a *dual-bus* (see figure 5.8). There are separate busses for packets leaving the sites and entering the sites, and there is a net-input/net-output pair connecting these two busses for intracluster communication. In terms of our earlier example, the transmission from *cluster-1* will complete, because the *input bus* for *cluster-2* is not busy, even though the *output bus* is.

The global bus, however, is implemented as a regular, single bus, as in figure 5.6.

To build a two-level bus, create a new design and go through same steps as for the single-level bus. The only difference is in specifying the bus structure—in step 2(a), choose a two-level bus. You will be prompted in the interaction window to specify the number of local busses and the number of sites connected to each local bus.

5.2.4 Bypassing the Editor

The *fastest* way of creating a new CARE design (of the same type as an existing one) is to simply transform edit the textual representation of the existing design. The function `create-new-design` is provided for that purpose.

The *required* arguments to `s:create-new-design` are:

new-design-name The name (without the `.x` extension) of the design file to be created.

existing-design-name The name of an existing design file.

The following are optional *keyword* arguments to `create-new-design`:

:existing-design-directory The directory containing the existing design file; defaults to the value of `s:*design-directory*`.

:new-design-directory The directory to contain the new design file; defaults to the directory specified by **:existing-design-directory**.

:dimensions For grid/torus circuits, this dotted pair specifies the number of columns and the number of rows, respectively, in the new design. For example, a value of `(3 . 4)` would create a grid/torus with 3 columns and 4 rows.

:number-of-busses For two-level busses, this specifies the number of *local* busses to create. A value of 1 *does not* create a single-level bus—it creates a two-level bus with one cluster!

:sites-per-bus For a single-level bus, this specifies how many sites are connected to the bus. For a two-level bus, this specifes how may sites are connected to each local bus.

If any of the parameters of the new design (such as **:dimensions**) are not specified, you are prompted to enter them, based on the type of the existing design. If the existing design is not a grid, torus, or bus, it cannot automatically generate a new design

Examples:

To create an octally-connected, 4 x 4 torus from an existing 2 x 2, type

```
(create-new-design 'my-octorus-16 'octorus-4 :dimensions '(4 . 4))
```

or type

```
(create-new-design 'my-octorus-16 'octorus-4)
```

and answer the prompts.

To create a two-level bus with four clusters of six sites each, from an existing 2 x 2 bus, type

```
(create-new-design 'my-bus-4x6 'bus-2x2
                   :number-of-busses 4
                   :sites-per-bus 6)
```

or type

```
(s:create-new-design 'mybus-4x6 'bus-2x2)
```

and answer the prompts.

5.3 Manual Wiring

The most general method of connecting components in P-HELIOS is to add *lines* to a design and manually route each line between two *ports*. Lines represent communication channels of unlimited width—any value may be sent across a line. Ports represent the communications interface of a component.

5.3.1 Example: Connecting Two Sites

1. Create a new design and define its viewing window.
2. Make the outer-level box big enough to hold two sites by zooming out (probably a factor of three is enough) and reshaping the bounding box to fill the viewing window.
3. Add two *Site* components, using the *Add Box* command. Use the default names for the sites, and place them anywhere you like. (The exercise will be more interesting if you don't place them exactly side-by-side.)
4. The little boxes around the sides of the sites are *ports*. In a CARE site, four of these ports make up a single bidirectional communications channel (two ports for data, two for status). We will be connecting the ports of the right-hand side of one site to the ports on the left-hand side of the other. Connecting two ports involves drawing a line between them.

 Select the *Add Lines* command from the *Edit Operations* menu; the mouse cursor will change to "×," and the following message appears in the interaction window:

   ```
   Choose the starting port (Middle button if done adding lines).
   ```

 Do the following sequence of steps for each pair of sites you want to connect. When done, click the middle button. (For this example, connect the top port on the right side of one site to the top port on the left side of the other, and so forth, as shown in figure 5.9.)

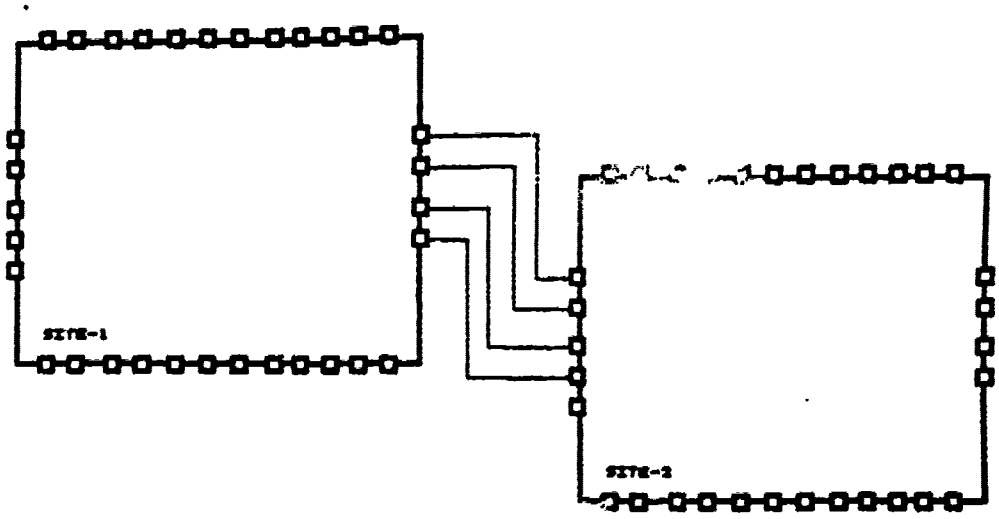


Figure 5.9 Manually wiring two sites together.

(a) Click left on the starting port. It will be highlighted—click right to confirm the selection.

(b) Choose the ending port in the same way, clicking left, then right.

(c) The interaction window prompt now says:

```
Define the line.
```

You may now specify an arbitrary zigzag path of horizontal and vertical line segments connecting the two points. Initially, the mouse cursor specifies a position in the horizontal direction[3]—click left to draw a line to that position. Moving the mouse and clicking left shortens or lengthens the line. Clicking right confirms this segment of the line, and allows you to route in the other direction.

(d) Repeat the above step until you reach the destination port, at which point the line has been added to the design.

(e) If you are finished adding lines, click the middle button. If not, return to step (a).

5.4 Semi-Automatic Wiring

The *array* component is an example of *semi-automatic* wiring in P-HELIOS. In this approach, you specify a component to be replicated in a two-dimensional pattern and manually wires the connections between one such component and its neighbors. The system then automatically replicates the component and its connections.

[3] Or the vertical direction, depending on the port orientation.

To build an array of components, you add an *Array* component to the design. The component to be replicated (e.g., a site) is added to the array component. You then specify the dimensions of the array—that is, the number of rows and columns—and the spacing between rows and columns.

P-HELIOS then displays the unit cell and "phantom" copies of its neighbors.[10] You then manually wire the unit cell to its neighbors.

When the array component is instantiated, it will be replaced by a *composite box* component which contains instantiated copies of the unit cell, connected in the specified pattern.

5.4.1 Example: Building an Array of Sites

In this example, we will build a quad-connected, 3 x 3 grid of sites. This is actually the method used to implement automatic wiring of grids described in section 5.2.

1. Create a new design and define its viewing window. Name it whatever you like.
2. Zoom out (a factor of three is probably enough), and make the bounding box larger. Also, set the display level to 2.
3. Add an *Array* component to the design. After it is placed, resize it (by *right*-clicking on *Reshape Bounding Box* in the *Edit Operations* menu and selecting the array component) to almost fill the outer-level bounding box.
4. Add a *Site* to the array (by *right*-clicking over the *Add Box* command and selecting the array component) and place it in the upper, left-hand corner. At this point, *four* sites will be displayed (probably drawn on top of one another, because the default spacing is too small). These correspond to the "typical" site and its nearest neighbors to the right and below. The neighbor sites are *phantoms*, as described above.
5. Select the *Modify Attributes* command from the *Edit Operations* menu, and select the array component. A menu will appear listing the attributes which may be specified.
 (a) Click on *Dimensions* in the attributes menu to specify the number of rows and columns in the array, for example:

   ```
   Number of columns:  3
   Number of rows:  3
   ```

 (b) Click on *Spacing* in the attributes menu to specify the inter-component spacing in the array. For example, the following represent good values for sites:

   ```
   Distance in the horizontal direction:  250
   Distance in the vertical direction:  200
   ```

[10] The neighbor cells are not actually instantiated in the representation of the array. They are merely displayed to aid you in specifying the connections between neighbors.

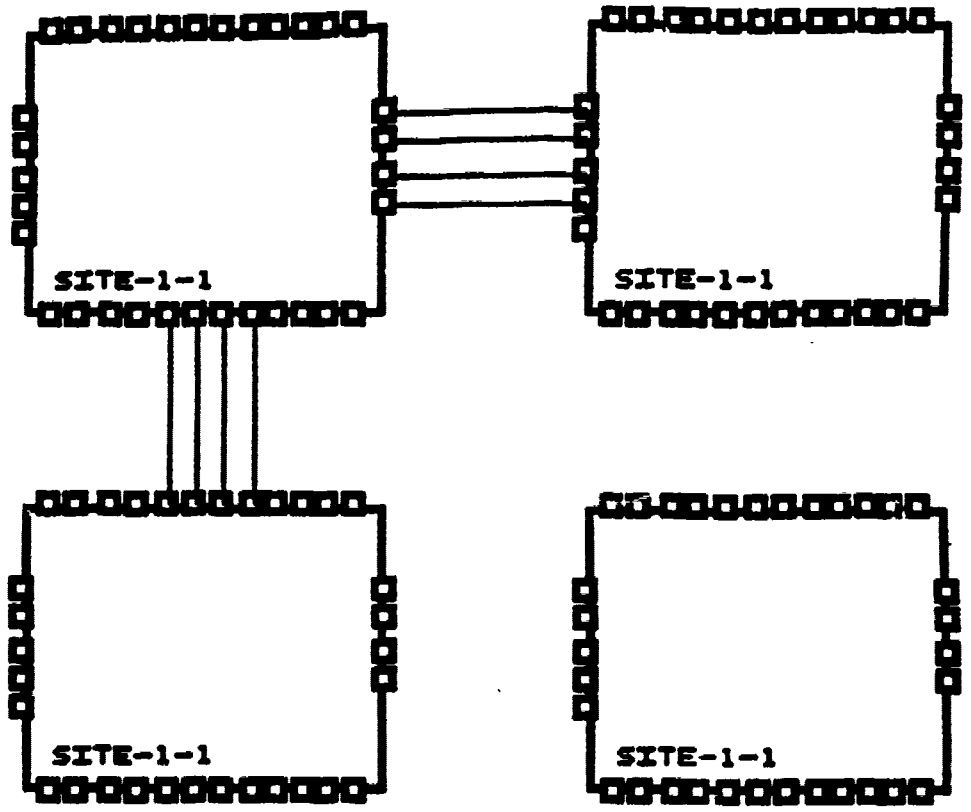


Figure 5.10: Wiring up neighbors in a CARE array.

(c) If this design is to be saved to a file, click on *Instantiate When Loaded* in the attributes menu, and type y in response to the prompt. This specifies that the design should be instantiated when the file is loaded.

(d) When finished modifying attributes, move the mouse away from the attributes menu, and it will disappear.

6. Select *Redraw Window* from the *Window Operations* menu (figure 5.4) to see the effect of the changes.

7. Next, manually wire (as in section 5.3) the connections between the upper, left-hand site and its neighbors, as shown in figure 5.10.

8. If the design is to be saved to a file, instantiate the site components[11] and then save the design. If you want to view (or edit) the design in its fully instantiated state, proceed to the following state. (This can also be done *after* the design is saved.)

9. Select the *Instantiate Box* command from the *Edit Operations* menu, and select the array. P-HELIOS will go busy for while, creating and wiring the new design. When "Done!" appears in the interaction window, the design has been instantiated. Use the commands in the *Window Operations* menu (figure 5.4) to view the design—you will need to redraw the window.

[11] SIMPLE doesn't know about libraries and box-descriptors and such, so "real" site must be saved, not a descriptor of one.

Chapter 6

Instrument Design

This chapter describes the instrumentation facilities available in SIMPLE/CARE. It is organized as three sections around the three key abstractions of the instrumentation system: probes, panels and instruments. Each of these is described in the context of what is available in CARE, followed by a description of SIMPLE facilities available to users interested in customizing these or in designing their own.

A CARE instrument is a facility to visualizes dynamically the internal state of a CARE (simulated) machine.[1] An *instrument* has a window which is divided into several regions called *panels*. Each panel displays a particular aspect of the state of a CARE design to which the instrument is attached. The information which is displayed in a panel comes from one or more *probes*. A probe is defined for a component of a CARE design and is responsible to monitor a particular aspect of the component.

6.1 Probes

This section describes the probes available with CARE as supplied. They may be used as a basis for specialization or for defining new probes.

Probes are the means by which useful data is extracted from a simulated design. In keeping with the SIMPLE design philosophy of 'partitioned concerns', probes are used primarily to collect (abstracted) state data in the design, leaving the aggregation and presentation of such data to the panels comprising the instrument.

Each probe is attached to a single component in the simulated (multiprocessor) design. During a simulation run, it is notified of the component's state changes and it uses these notifications to collect state data about the component. A probe monitors some particular aspect of a component, hence, it is not unusual to have a number of probes (of different types) monitoring the same component. Each probe is also attached to one or more panels and provides these with the abstracted state data about the probed component during the simulation run.

[1] We often use a term *design* for a specific CARE machine

Probes are implemented as flavor instances. Each type of probe is associated with a specific type of component. Each probe type also has associated with it a *probe key* keyword, allowing data from a probe of this type to be identified as such by any panels associated with the probe.

6.1.1 The Value Passing Measurement Model

CARE provides a family of probe types to monitor system components in the modeled value passing machine. These monitor:

- The *status* of various components in the design. As components alternate between a busy state and a free state, CARE status probes report these state changes, mapping them into 1 and 0 respectively.
- The *queue* on various components in the system. CARE probes report the sum of the lengths of the queues providing work for that component.[2] For example, the queue of work for an evaluator is the number of pending active **care-processes** waiting to be run, that is, the queue length associated with its **Evaluator-Queue** instance variable.
- The *latency* experienced by application tasks at various stages of processing. The latencies reported by CARE probes in the value passing model are based on the LAMINA computation model of asynchronously communicating objects. Recall that in this model application work is accomplished by objects passing request messages on task streams. These messages experience various latencies before the requested task is actually accomplished. The names used by the probes to refer to these latencies are detailed in figure 6.1, and are explained below.

Latencies in the Value Passing Model

The *source delay* or *launch delay* is the interval between a LAMINA object initiating a request in the evaluator to send a message (via sending, for example), and the source operator (the operator at the sending site) passing it to the network for transmission to the target site. This measure incorporates both the time the message spends waiting for operator attention as well as the time the operator takes in performing the service (interrupt, and formatting the message into a packet).

The *net delay* is the time the packet containing the request message takes to traverse the network—from the time the source operator hands it to the network to the time it arrives completely in the input buffer at the target site.

Once a packet arrives at the target site's input buffer, it usually spends some time waiting for the target operator's attention. This latency is referred to the *operator queueing delay*. Note that this latency is defined only for arrivals from the network—locally targeted packets never enter the input buffer. Thereafter, there is a latency associated with the target operator servicing the packet (interrupt, packet decoding, queueing the packet on the target stream, and perhaps enabling **care-processes** waiting on that stream); this is the *operator service delay*. Together, the queueing delay and service delay are sometimes referred to as the *operator delay*.

[2] Panels usually transform this integer into a percentage.

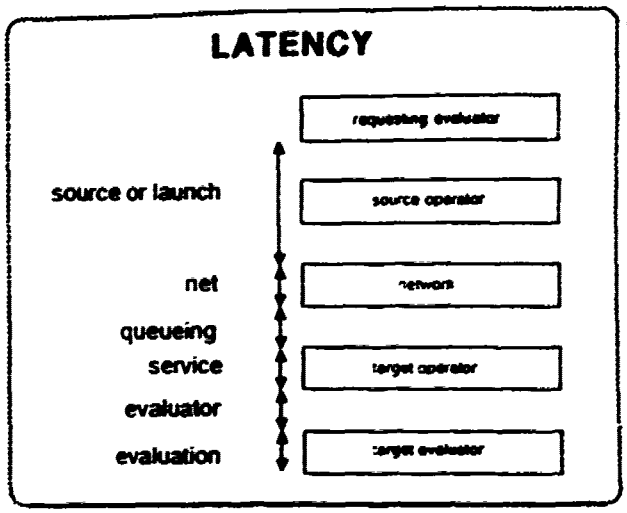


Figure 6.1: Latencies

Once a care-process that was awaiting the arrival of a message on a stream is enabled, it is passed to the target evaluator. Here, the time it spends in the queue of runnable processes is called the *evaluator delay*. Thereafter, the runtime of the task is called the *evaluation*.

The following section describe each existing probe grouped by the type of a component to which the probe is attached.[3]

6.1.2 Evaluator Probes

The following probes attach to evaluator components.

Evaluator-Status-Probe : Reports on the status of the associated evaluator.

Probe key:	:evaluator-status
Probe object:	The enclosing site component.
Update items:	
:new	Current status: 1 or 0.
:last	Previous status: 1 or 0.

Evaluator-Queue-Probe : Reports on the queue of the associated evaluator.

Probe key:	:evaluator-queue
Probe object:	The enclosing site component.
Update items:	
:busy	A count of the pending care-processes in Evaluator-Queue, plus 1 if evaluator is in a busy state.

Evaluator-Latency-Probe : Reports on the latencies associated with the care-process that was just evaluated by the associated evaluator.

Probe key:	:evaluator-latency
Probe object:	Object in the context slot of the care-process that was evaluated (usually a LAMINA object).
Update items:	
:process	The care-process that was evaluated.
:stream	The stream that provided the above care-process with work.
:process-context	Object in the context slot of the care-process that was evaluated (usually a LAMINA object).
:process-class	The type of the above object.
:stream-queued	The number of packets remaining on the above stream.
:launch-delay	Simulated μs. See section 6.1.1.
:net-delay	Simulated μs. See section 6.1.1.
:operator-delay	Simulated μs. See section 6.1.1.
:evaluator-delay	Simulated μs. See section 6.1.1.
:evaluation	Simulated μs. See section 6.1.1.

[3]The names of all defined probe types can be found in the global variable s:*all-probe-names*.

Process-Activity-Probe : Reports on the activity of the process running in the evaluator. This probe type is specific to reference passing CARE systems.

Probe key:	**:process-activity**
Probe object:	The function in either the **tag** or the **context** slot of the `care-process`.
Update items:	
:process	The `care-process` running in the evaluator.
:process-context	The function in either the **tag** or the **context** slot of the `care-process`.
:process-class	The name of the above function.
:evaluation	The simulated μs (last) spent executing without making a request to shared memory.
:busy-wait	The simulated μs (last) spent waiting for a request to shared memory to complete.

Node-Queue-Probe : Reports on the queue load[4] of an evaluator which represents the specific type of `care-processes` queued for an evaluator to process.

Probe key: *Probe object*:	**:node-queue** The enclosing site component.
Update items: **:node-queue**	The queue load of a specific type of `care-process` to be processed by an evaluator.

[4] The queue load of a component is defined as the number of items in queues associated with the component plus one if the component is busy processing an item.

6.1.3 Operator Probes

The following probes attach to **operator** components.

Operator-Status-Probe : Reports on the status of the associated operator.

Probe key:	**:operator-status**
Probe object:	The enclosing site component.
Update items:	
:new	Current status: 1 or 0.
:last	Previous status: 1 or 0.

Operator-Queue-Probe : Reports on the queue of the associated operator.

Probe key:	**:operator-queue**
Probe object:	The enclosing site component.
Update items:	
:busy	The sum of the number of (1) packets in the input buffer from the network, (2) packets from the evaluator in **From-Evaluator-Queue**, and (3) locally targeted packets in **Local-Packet-Queue**, plus 1 if the operator is in a busy state.

Operator-Latency-Probe : Reports on the latencies associated with packets arriving at the associated operator from the network.

Probe key:	**:operator-latency**
Probe object:	The enclosing site component.
Update items:	
:launch-delay	Simulated μs. See section 6.1.1.
:net-delay	Simulated μs. See section 6.1.1.
:queueing-delay	Simulated μs. See section 6.1.1.
:service-delay	Simulated μs. See section 6.1.1.

6.1.4 Network Probes

This section describes the probes that are relevant to network activity. They attach to **net-output**, **fb-in** and **fb-out** components.

Net-Output-Connection-Probe : Signals the path of a packet as it is routed through a site.

Probe key:	**:net-output-connection**
Probe object:	The **net-output** component.
Update items:	
:status	**:open** or **:free**.
:points	A list of the points are given in terms of the x - y position of connecting elements as defined by the designer's interaction with the structure editor.

Network-latency-Probe : Reports on the net latency of a packet which arrives at this site successfully.

Probe key:	**:network-latency**
Probe object:	The **fb-in** component.
Update items:	
:delay	The net delay which the packet has experienced.
:total-size	The size of the packet in (32 bit) words.

Offered-Load-Probe : Reports on the load "offered" to the network by the associated operator through the associated fifo buffer, **fb-out** (i.e., the number of targets in packets as they are launched into the network from the associated operator). This probe is attached to a **fb-out** component.

Probe key:	**:offered-load**
Probe object:	The site component enclosing the fb-out.
Update items:	
:load	The number of distinct targets in the packet just launched by the fb-out, indicating load offered to the network.

6.1.5 Bus Probes

This section describes the probes that are relevant to bus activity. They attach to **bus-mixin**, **net-output**, and **net-input** components.

Bus-Queue-Probe : Indicates when the bus is busy.

Probe key:	**:operator-queue**
Probe object:	The **bus** component.
Update items:	
:busy	1.0 if the bus is used, 0.0 otherwise.

Bus-Output-Connection-Probe : Signals the path taken by a packet sent to the bus.

Probe key:	**:bus-connection**
Probe object:	The **net-output** component.
Update items:	
:status	**:open** or **:free**.
:points	A list of the points on the path taken by the packet to the bus. See the explanation of **:points** item for **net-output-connection-probe** in this section.

Bus-Input-Connection-Probe : Signals the path taken by a packet send from the bus.

Probe key:	**:bus-connection**
Probe object:	The **net-input** component.
Update items:	
:status	**:open** or **:free**.
:points	A list of the points on the path taken by the packet from the bus. See the explanation of **:points** item for **net-output-connection-probe** in this section.

6.1.6 Defining and Specializing Probes: Defprobe

This section describes the interface provided by SIMPLE to define new probe types, and discusses the essentials of what this requires.

Probe's :Trigger Method

The first filtering of events for the purposes of instrumentation is done by probes. Some events cause no further measurement activity since not all events merit action by the particular instrument. The decision is made and information is collected in the **:trigger** method of the probe, to which is available:

- the event, which is passed in to the method as its parameter; the **with-event-bindings** macro is used to destructure the event around the body of code:

 with-event-bindings *event time* **&body** *body*

 The *event* is the same name used to denote the event parameter of the method (for example, **s:event**), and *time* is the variable name used to identify the time of the event within the code (for example, **now**). The lexical variables **s:event-object**, **s:event-slot-name** and **s:event-slot-value** are bound by the macro to their appropriate values.[5]

- the ports and state variables of the probed component, through the macros **probe-via** and **probe-state**.

- the state variables of the probe.

Each piece of selected information is then tagged with an identifying keyword and collected into a disembodied property list. This list is passed along as part of the **:update** message to the connected panels, along with the probe key, the probed object and the simulated time (see section 6.2.6 for the function of the **:update** method). The probed object might be the probed component, some other component related to it in some way (for example, the enclosing site), or some data structure manipulated by it (for example, a LAMINA object).

A probe may be composed of predefined mixins to do standard calculations (for example, a time weighted average). SIMPLE facilitates this by establishing a message protocol that is followed by all probes. Thus, the **:trigger** method invokes the **:calculate** method of the probe, which, in turn, invokes its **:select** method, which, finally, invokes the **:update** method of the selected panels associated with the probe. A probe is composed by naming it as a specialization of appropriate mixins, which undertake to shadow these messages with their own methods. The default behavior is to pass information through without change to all panels connected to the probe (see section 6.1.8 for the functions of **:calculate** and **:select** methods).

[5] The behavior predicates like **state-event** and **via-event** require the use of **with-event-bindings** around the code that uses them. The current implementation of CARE requires that the symbol **s:event** to be used to name the *event* parameter, and that the symbol **now** be used to denote the time.

Definition of Defprobe

A new probe type is needed when either (1) the information provided by existing probes for a component type is inappropriate or (2) a new component type has been designed. The **defprobe** macro is provided by SIMPLE to accomplish this task.

defprobe *name* (**&key** *key-args*) **&body** *trigger-method*

defprobe defines a flavor named *name* and the **:trigger** method for the flavor (if specified). It also defines a **:before :reset** method to initialize probe instance variables (if specified) when a simulation is reset.

name is an unquoted symbol identifying the type of probe being defined and the flavor of this name is to be created.

The keyword arguments usable in *key-args* are:

:component-type is an unquoted symbol identifying the type of component, (for example, **c:evaluator**). If left unspecified, this is presumably inherited via *mixins*.

:probe-key is an unquoted keyword symbol that will serve to identify the data from an instance of this probe within a panel.

:ivs is a list of instance variables for the probe flavor, along with their initializations if any.

:mixins is an unquoted list of flavor names (types) which this probe type will specialize. Any element of the list may be a probe defined by **defprobe**.

:documentation is a documentation string for the probe type.

The *trigger-method* body is also optional. It declares the code for the probe's **:trigger** method. If it is left unspecified, the method is inherited, presumably via some mixin.

6.1.7 Example Probe Definitions

This section shows the definition of a probe for CARE, evaluator-queue-probe.

```
(defprobe EVALUATOR-QUEUE-PROBE
  (:component-type evaluator
   :probe-key :evaluator-queue
   :ivs (input-queue)
   :documentation "Report evaluator status")
  ;; Define the :trigger method.  Parameter = s:event.
  (with-event-bindings s:event now  ; destructure event, bind time, etc.
  ;; with-event-bindings binds this but it is not used
    (ignore s:event-slot-value)
    ;; status changed?  process arrived from operator?
    (when (or (state-event status) (via-event packet-in))
      (probe-calculate     ; i.e. (send self :calculate ...)
        :evaluator-queue  ; probe key
        (list :busy
              (+ (internal-queue-length input-queue evaluator-queue)
                 (case (probe-state status)
                   ((ready busy-wait) 0) (otherwise 1))))
        owning-box))))     ; probed object = evaluator's enclosing site
```

The predicates **state-event** (and **via-event**) check that their argument is identical to the value of **s:event-slot-name** (bound by **with-event-bindings**). Note the use of **probe-state** to determine the current value of the evaluator's instance variable **Status**. The macro **internal-queue-length**, as used, sets the probe's own instance variable **Input-Queue** to the value of the evaluator's instance variable **Evaluator-Queue** and returns the queue length of this value. The macro **probe-calculate** sends **self** the **:calculate** message with the given parameters, after converting the time which is the value of **now** from event units to simulated microseconds.

6.1.8 Probe Details

Basic Probes

All component probe flavors have **s:basic-probe** as their base flavor, which provides the following instance variables:

- **s:probed-component**: The type of component to which the probe is attached (for example, an evaluator). This is set and used by SIMPLE.
- **owning-box**: The supercomponent of the probed component (for example, a site). This is set and used by SIMPLE.
- **s:panels**: The list of panels to which the probe is connected. This is set and used by SIMPLE.
- **s:filter**: A predicate to be applied to the probed object, the probe key, the update items and the simulated time, within the body of the default **:calculate** method. A `nil` returned value disables the **:select** message from being sent to the probe's connected panels. Changing this allows for easy specialization of probe behavior.
- **s:selector**: A predicate applied to each component to determine whether a component probe of this type must be attached to it. This is set and used by SIMPLE (see section 6.2.5 for the usage of this instance variable).

:Trigger, :Calculate, and :Select Methods

After a probe receives a **:trigger** message as a result of a state change in the connected component, a **:calculate** message is sent to itself. The main function of **:calculate** method is to provide an extra facility to manipulate data before they are sent to panels. There is, however, another useful function which can prevent data from being sent to panels at all by providing the **s:filter** instance variable of the probe with an appropriate predicate function. The **:calculate** method, if not blocked by the **s:filter** predicate, sends a **:select** message itself at last with possibly modified arguments which have received.

The **:select** method sends a **:update** message to each of the panels connected to the probe with the same arguments as received.

The **s:basic-probe** defines stub methods for the **:trigger**, **:calculate**, and **:select** messages which any probe can specialize.

Template Probes

When an ,trument is created, it has a list of template probes associated with it. There is one template probe i nce for each unique probe type required by the panels of the instrument. It is the task of the template probes to attach probe instances of a specific type to the components of the design, and to reset

these whenever the simulation is reset. In addition, the template probes carry out sundry housekeeping tasks, like suspending and resuming probe operations.

6.2 Panels

This section describes the panels available in CARE and the SIMPLE interface for users to define new panels or specializing existing panels.

Panels are the means by which data passed by associated probes are displayed on the screen. Panels may be categorized into several types in terms of their graphical *presentations*. Presentation types are implemented as flavors, and panels are built upon the presentation flavors with some of their instance variables, such as probes, specialized.

6.2.1 Panel Presentation Types

SIMPLE provides various presentation types which are divided broadly into five groups.

Presentation Groups	*Presentation Types (flavors)*
Box and line presentations	**Bal-presentation**
Histogram presentations	**Histogram-plot-presentation** **Time-histogram-plot-presentation** **Dual-histogram-plot-presentation** **Time-dual-histogram-plot-presentation**
Scrolling pattern presentations	**Scrolling-bar-plot-presentation**
Point and line presentations	**Point-plot-presentation** **Line-plot-presentation** **Scrolling-line-plot-presentation**
Text presentations	**Scrolling-text-presentation** **Lisp-listener-presentation** **Truncation-lisp-listener-presentation** **Text-presentation**

1. The *box and line presentations* deal with data in terms of the descriptive picture of a simulated design. It displays two different kind of objects; lines and boxes. The typical usage of **Bal-presentation** is that it shows network activities in terms of lines drawn between boxes and the activities of a box in terms of a color or a gray shade drawn inside the box.[6]

2. The *histogram presentations* deal with data displayed in the form of histogram. There are two kinds of histogram presentations available in SIMPLE. One is an ordinary type, **Histogram-plot-presentation**, which simply counts the occurrences of an event represented by each histogram bin. The other, **Time-histogram-plot-presentation**, keeps track of time spent by a state represented by each bin. The both types of histogram presentations may have two histogram regions which share the x axis. Those are called **Dual-histogram-plot-presentation** and **Time-dual-histogram-plot-presentation** respectively.

3. The *scrolling bar presentations* deal with data displayed in the form of colored or shaded bars which scrolls over time as the x axis typically represents the simulated time. A panel of this group displays an

[6] Colors are used on a color monitor while gray shades are used on a black and white monitor.

aspect of some specified objects. **Scrolling-bar-plot-presentation** has its screen divided horizontally into a specified number of rectangular bars. Each bar is responsible of keeping track of the states of one or more specified objects (see *aggregation* item in the table in section 6.2.1).

4. The *point and line presentations* deal with data displayed in the form of *points* and *lines*. A presentation type of this group has x and y axes. X-axis is called **bottom-axis** and there are two y axes; one on the left side is called **left-axis** and the other on the right side is called **right-axis**. The **right-axis** may not be used. **Point-plot-presentation** displays points (dots) in the x-y plane. **Line-plot-presentation** displays lines (curves). **Scrolling-line-plot-presentation** displays lines which scrolls along the x axis which usually represents the simulated time.

5. The *text presentations* deal with data displayed in the form of text. **Scrolling-text-presentation** displays lines of text sorted in a specified order. **Text-presentation** simply displays user supplied text and does not deal with data from a probe. **Lisp-listener-presentation**, as well, does not interact with any probe, and it is a lisp listener which a user can use to evaluate lisp expressions. **Truncation-lisp-listener-presentation** is similar to the **Lisp-listener-presentation** except that if one line of text is too long to fit in the panel region, the line is truncated.

Mouse Changeable Panel Attributes

By clicking a right button of a mouse over the region of a panel, you can change some attributes of the panel which are specific to the type of the presentation.[7] The following table summarizes the available attributes and the presentation types which provide the attributes.

Attribute Names	*Presentation Types*
Display Interval	all presentation types except **Bal-presentation**, **Text-presentation**, **Lisp-listener-presentation**, and **Truncating-lisp-listener-presentation**.
Sampling Interval	All presentation types except **Bal-presentation** and all presentation types in *text presentation* group.
Scroll Range	**Scrolling-line-plot-presentation** and **Scrolling-bar-plot-presentation**.
Text font	All presentation types in *text presentation* group.
Normalized	All presentation types in *histogram presentation* group.
Bin Size and Overflow threshold	All presentation types in *histogram presentation* group.
Aggregation	**Scrolling-bar-plot-presentation**.

- The *display interval* controls the frequency of refreshing the contents of a panel. It is specified in real-time seconds or `nil`.[8] Any attempt of displaying operation is ignored unless more than display interval value of time has elapsed since the last time the panel has refreshed its contents.

- The *sampling interval* controls the frequency of incorporating **:update** messages from probes to construct the contents of the panel. By supplying non zero value to the sampling interval of a panel,

[7] The mouse changeable attributes of a panel are usually defined as instance variables of its presentation type.
[8] With `nil` all attempts of displaying operations are processed.

simulation may be speeded up as some of the **:update** messages are simply ignored, but the display may not reflect exactly what has happened in the simulation. It is specified in simulated microseconds or **nil**.[9] An **:update** message is ignored unless this is the first message about the probed object or more than sampling interval value of time has elapsed since last time the panel has processed an **:update** message.

- The *scroll range* determines the range of the x axis of a scolling plot presentation.
- The *text font* controls a font used to display text. You can change it by choosing one from a list of available fonts.
- The *normalized* controls histogram display format, either normalized or not. If it is normalized, the heights of the histogram bins are marked between 0 % and 100 % by normalizing values over all the occurrences in the case of **(dual-)histogram-plot-presentation** or over total time in the case of **time-(dual-)histogram-plot-presentation**.
- The *bin size and overflow threshold* controls the size of the bin and the threshold value of the overflow bin if an overflow bin is used for any histogram presentation panel.
- The *aggregation* controls the aggregation of bars for **scrolling-bar-plot-presentation**. For example, each bin can represent the average state of two objects by specifying two for the aggregation. You can change the aggregation to any value as long as the height of the panel is big enough to display all the bars.

Panel Facilities Invoked by Mouse

Some presentations have some facilities which are invoked by mouse actions. As explained in the previous section, a right click over a panel screen is reserved for the purpose of changing the attributes specific to the panel type. A middle click over any panel, i.e. over an instrument window, is also reserved and explained in section 6.3.3. Currently two types of presentations have special facilities.

- Boxes displayed in a **bal-presentation** panel are mouse sensitive. A left click over one of the boxes invokes an inspector window to inspect the box. A right click over a box gives you a menu of available operations on the box, but there is currently just one item on the menu which is "inspect" and has the same effect as a left click over it.
- Lines displayed in a **scrolling-text-presentation** panel can be mouse sensitive. If the instance variable **mouse-sensitive-line-p** has non-nil value, then clicking on a line invokes an inspector window to inspect the last item listed in the **text-items-form** instance variable of the panel.

[9] **nil** means no sampling, i.e. all the **:update** messages the panel receives are processed

6.2.2 Available Panels

A panel which is used in an instrument should be defined by the SIMPLE construct **defpanel** (see section 6.2.5 for the definition of the **defpanel**). We use term *presentation* to indicate a flavor of a presentation type and *panel* to indicate anything defined via **defpanel**. Thus, a panel is built upon a presentation. The names of all panels currently available are listed below with regard to the underlying presentation types.[10]

Presentation Group	*Panel Names*
Box and line presentations	**Net-operator-qload-site-mapping-panel** **Processor-and-memory-qload-mapping-panel** **Bus-operator-qload-mapping-panel** **Node-queue-logarithmic-site-mapping-panel**
Histogram presentations	**Evaluator-operator-histogram-panel** **Processor-memory-histogram-panel**
Scrolling pattern presentations	**Evaluator-qload-scrolling-bar-panel** **Processor-qload-scrolling-bar-panel** **Memory-qload-scrolling-bar-panel**
Point and line presentations	**Cumulative-latency-panel** **Evaluator-queue-history-panel** **Processor-queue-history-panel** **Network-latency-panel** **Network-offered-load-latency-panel** **Operator-potential-latency-panel** **Evaluator-potential-Latency-Panel**
Text presentations	**Instance-activity-panel** **Class-activity-panel** **Lisp-panel** **SV-lisp-panel** **Notes-panel**

Net-operator-qload-site-mapping-panel :

Connected probes: **net-output-connection-probe**, **operator-queue-probe**.

This panel shows the network activity in line segments and the queue load of operators of sites in painted boxes. The network wire is drawn if it is used to transmit a packet.

Processor-and-memory-qload-mapping-panel :

Connected probes:

net-output-connection-probe, **operator-queue-probe**, **evaluator-queue-probe**.

This panel is a shared memory design version of the **net-operator-qload-site-mapping-panel** in the sense that a box pattern shows the queue load of either the processor or the memory depending on the function of the box.

Bus-operator-qload-mapping-panel :

Connected probes:

bus-output-connection-probe, **bus-input-connection-probe**, **bus-queue-probe**.

[10] The names of all defined panels can be found in the global variable s:*all-panel-names*.

This panel is a bus design version of the **Net-operator-qload-site-mapping-panel**. It shows the network activity of the bus.

Node-queue-logarithmic-site-mapping-panel :

Connected probes: **net-output-connection-probe**, **node-queue-probe**.

This panel is similar to the **Net-operator-qload-site-mapping-panel** except that the box pattern shows the load for a specific type of **care-processes** in the evaluator.

Evaluator-operator-histogram-panel :

Connected probes: **evaluator-status-probe**, **operator-status-probe**.

This panel has two histogram regions sharing the x axis. The x axis is divided into bins each of which represents the number of sites if the width of the panel is big enough to display all the bins. Otherwise some aggregation may be made over the x axis. For example, if the panel is not wide enough to display N bins where N is the number of sites in a design, but is wide enough to display $N/2$ bins, then n-th bin may be used to represent both $2n$ sites and $2n+1$ sites. In the case of no aggregation used, the height of the n-th bin in the upper histogram indicates the portion of time n evaluators have been busy while that in the lower histogram indicates the portion of time n operators have been busy.

Processor-memory-histogram-panel :

Connected probes: **evaluator-status-probe**, **operator-status-probe**.

This panel is a shared memory design version of the **Evaluator-operator-histogram-panel**. The upper histogram displays the utilization of processor site while the lower histogram displays the utilization of memory site.

Evaluator-qload-scrolling-bar-panel :

Connected probes: **evaluator-queue-probe**.

This panel associates a bar to each site if the panel height is big enough to display all the bars. Otherwise some aggregation is performed over the sites (see *aggregation* item in *Mouse Changeable Panel Attributes* paragraph of section 6.2.1). Each bar shows the transition of the queue load of the evaluator of the associated site in colored or shaded rectangles over time.

Processor-qload-scrolling-bar-panel :

Connected probes: **evaluator-queue-probe**.

This panel is similar to the **Processor-memory-histogram-panel** except that it is used on a shared memory design and shows the queue load of the processor.

Memory-qload-scrolling-bar-panel :

Connected probes: **operator-queue-probe**.

This panel is similar to the **Processor-qload-scrolling-bar-panel** except that it shows the queue load of the memory.

Cumulative-latency-panel :

Connected probes: **evaluator-latency-probe**.

This panel shows five curves of the latencies which process contexts in evaluators have experienced The five curves indicate:

- launch delay,
- this last plus net delay,
- this last plus operator delay,
- this last plus evaluator delay, and

- this last plus evaluation delay

of each process context[11]. Note the five curves are strictly cumulative. Thus they may overlap each other but may not cross each other. The x axis indicates the "rank" of each process context. A process context has a "rank" assigned in decreasing order of the total delays for its most recent invocation. The ranks are recalculated every time the panel refreshes its contents.

Evaluator-queue-history-panel :
Connected probes: **evaluator-queue-probe.**
This panel is an x-y plot panel with a scrolling x axis which indicates the simulated time. The curve to be drawn shows the sum of queue lengths of all the evaluators.

Processor-queue-history-panel :
Connected probes: **evaluator-queue-probe.**
This panel is similar to the **Evaluator-queue-history-panel** except that it shows the sum of processor queue lengths.

Network-latency-panel :
Connected probes: **operator-latency-probe.**
This panel is a scrolling line plot panel which shows the net delays of packets observed over time.

Network-offered-load-latency-panel :
Connected probes: **operator-latency-probe, offered-load-probe.**
This panel is a scrolling line plot panel with the both sides of y axis utilized. The left y axis shows the total network load offered by all the operators while the right y axis shows two latency curves; the net delay and the sum of the net delay and the launch delay of a packet which has arrived at an operator.

Operator-potential-latency-panel :
Connected probes: **operator-queue-probe, operator-latency-probe.**
This panel is similar to the **Network-offered-load-latency-panel** except the left axis shows the two curves about potential operators while the right axis shows the two curves of operator latencies. One of two curves of potential operators indicates the number of operators whose queue length is less than 3 but not 0, and the other indicates the number of operators whose queue length is equal to or longer than 3. The two latency curves are the service delay and the sum of the queueing delay and the service delay of a packet which has arrived at an operator.

Evaluator-potential-latency-panel :
Connected probes: **evaluator-queue-probe, evaluator-latency-probe.**
This panel is similar to the **Operator-potential-latency-panel** except that the two potential curves shows the numbers of potential evaluators and the two latency curves shows the evaluation time and the sum of the evaluation time and the evaluator delay of a packet.

Instance-activity-panel :
Connected probes: **evaluator-latency-probe.**
This panel is a scrolling text panel which shows the activity of process contexts. Each line of text represents the activity of a context *instance*. One line consists of seven items:

- the average expected service time needed for the context in milliseconds[12],
- the stream queue length,

[11] See figure 6.1 for the meanings of the various latencies
[12] This value is calculated by multiplying the average service time for one activation by the stream queue length

- the average service time for one activation of the context in milliseconds,
- the number of total context activations,
- the *launch* + *net* + *operator* + *evaluator* latency of the current packet,[13]
- the CARE site of the context,and
- the name of the context.

Process contexts, and thus text lines, are ordered in decreasing order of averaged expected service time, and if the expected service times are approximately equal, in decreasing order of the latency associated with the most recent invocation of the object.

Class-activity-panel :

Connected probes: **evaluator-latency-probe.**

This panel is similar to the **Instance-activity-panel** except that each line represents the class of a process context. If a process context is a simple function object, then the class of the process context is defined to be the name of the function object. If a process context is a method invocation for an object, such as in an object oriented programming, the class of the process context is defined to be a pair of the object class and the method name. Each line consists of six items;

- the expected service time needed for a context of the class averaged over all the instances of the class in milliseconds,
- the average stream queue length,
- the average service time for one activation of a context of the class,
- the number of total activations,
- the number of all instances of the class, and
- context class as defined above.

Lisp-panel :

Connected probes: None.

This panel is just another lisp listener that a user can use to evaluate any lisp expression.

Notes-panel :

Connected probes: None.

This panel is a text presentation panel which does not associate itself with any probe. It is useful when user simply wants to display some comments about the simulation somewhere on the screen

.4 Interface Specifications

Before we start talking about how to customize panels, we need to explain a common record type which contains all the information necessary for a panel to handle data from a probe The record type is called the **panel-interface-spec** and a panel keeps one or more objects of this record type in the instance variable **interface-specifications** [14] Interface specification objects of a panel are created by processing a special

[13] See figure 6.1 for the meanings of the various latencies

[14] The number of the interface specification objects is determined by the presentation type of a panel. Most of the presentation types have only one interface specification while some of the presentations have two of them as they monitor two fairly independent aspects of a design. Among the currently available presentation types only the bal-presentation and the (time)-dual-histogram-plot-presentation have two.

set of instance variables which are provided by the presentation type of the panel. We call such an instance variable an *interface instance variable* (see section 6.2.4 for various interface instance variables). The actual procedure of customizing a panel interface is performed by specializing some of the interface instance variables of the panel.

```
(defstruct (panel-interface-spec (:conc-name panel-) :named)
  (states-arrays-size 1)
  (states-arrays-prototype nil)
  (states-arrays-resource-list nil)
  (states-arrays-hash (make-hash-table :size 50.))
  (states-arrays-updated (make-generic-queue))
  (states-arrays-display-hash (make-hash-table :size 50.))
  (update-closure '())
  (slot-save-functions '())
  (analysis-closure '())
  (display-operations '()))
```

When a panel receives data from a probe and it decides to process the data, then the data is encoded into arrays which we call a *states array* and a *display array*. A states array is used to keep a state value of a probed object. A reason why we keep a display array, separate from a states array, is that some panel may want to modify a states array to meet a specific displaying goal and also keep an original states array as it is to keep track of states of each object. Display arrays are queued in the record element, **states-arrays-updated**, until they are processed to be displayed on the screen. The arrays are reused for the sake of better performance by keeping a list of available arrays in the element, **states-arrays-resource**.

Two hash tables, **states-arrays-hash**, **states-arrays-display-hash** can be utilized to customize the update method of a panel. One is meant to be used to organize a *states array* which contains the current or last status of each object while the other is to organize a *display array*.

The **update-closure** element provides a closure which determines the major part of the **:update** behavior. The value of the **update-closure** is set by parsing an appropriate *update form* instance variable provided by the presentation type of the panel (see section 6.2.4 and section 6.2.7 for customizing update forms). The function of the **update-closure** is first to extract appropriate information from probe data and save it to a states array and a display array, and second to queue the display array to the **states-arrays-updated** queue. The first function for an interface specification is further defined in the **slot-save-functions** element to be described shortly.

The **analysis-closure** element provides a closure which is meant to be executed to manipulate display arrays just before the panel refreshes its display contents. The value of the **analysis-closure** is set by parsing an appropriate *analysis form* instance variable provided by the presentation type of the panel (see section 6.2.4 and section 6.2.7 for customizing analysis forms).

The values of the **slot-save-functions** and **display-operations** elements are set by parsing the special interface instance variables[15] defined for the presentation type of the panel. The **slot-save-functions**

[15] They are called *transformation item interface instance variables*. See section 6.2.4.

provides the functions by which each element of a states array and a display array is set. The **display-operations** element determines how to construct the intermediate data structure which is created from a list of display arrays and which the presentation type of the panel uses to refresh its display.

6.2.4 Interface Instance Variables

Interface instance variables need to be set properly in corresponding initialization forms of the **defpanel** definition of a panel in order to make the interface to appropriate probes work correctly. They are used to customize a panel.

There is one interface instance variable which is common to all type of panels. It is called **probes** and represents types of all the probes which may send data to the panel. The **probes** instance variable needs to be set to a list each element of which represents one kind of probe and consists of at least two items; the first item is a keyword which is used to identify the probe type in any transformation item interface instance variable as explained shortly, the other is the name of the probe type. The rest of the items are alternating `init` keywords and their values for the type of probe.

The other interface instance variables are categorized into three groups. One is an *update form* interface instance variable which is a form to be compiled into the **update-closure** of an interface specification record. The second is an *analysis form* interface instance variable which is a form to be compiled into the **analysis-closure**. The last is called a *transformation item* interface instance variable which determines what kind of data is extracted from a probe and how it is processed for display. For each interface specification record of a panel there exists a unique set of an update form interface instance variable and an analysis form interface instance variable.

Update Form and Analysis Form Interface Instance Variables

An *update form* instance variable contains a form which determines the behavior of the panel for an instance specification record when the panel receives an **:update** message from a probe.[16] The value of an update form is processed and compiled into the **update-closure** element of a corresponding interface specification record.

An *analysis form* instance variable is meant to specify an extra step of processing data received from a probe. The analysis form is processed and compiled into the **analysis-closure** element of a corresponding interface specification record. Any existing presentation type executes its compiled analysis form just before the panel displays the data, but a user can define a new presentation type which invokes it any time he wishes.[17]

- The **bal-presentation** deals with two interface specification records, one for line drawing, and the other for box drawing. There is one set of an update form instance variable and an analysis form instance variable for each instance specification record. The instance variables are called **line-update-form**, **line-analysis-form**, **box-update-form** and **box-analysis-form**.

[16] If **update form** is neither specified in the definition of a panel nor inherited from components panels, it defaults to '(send **self** :update-states-array).

[17] The invoking of the analysis form is done by using the SIMPLE construct perform-analysis-operation

- Each of the **histogram-plot-presentation** and the **time-histogram-plot-presentation** types has one interface specification record specifying **histogram-update-form** and **histogram-analysis-form** instance variables.

- Each of the **dual-histogram-plot-presentation** and the **time-dual-histogram-plot-presentation** has two interface specification records one of which is equivalent to the one for the the **histogram-plot-presentation** and the **time-histogram-plot-presentation**. In addition to the records specified by the **histogram-update-form** and the **histogram-analysis-form**, they have **lower-histogram-update-form** and **lower-histogram-analysis-form** instance variables to specify the other record.

- The **scrolling-bar-plot-presentation** and any type of point and line presentations have one interface specification record and have **plot-update-form** and **plot-analysis-form** instance variables to specify this record.

- The **scrolling-text-presentation** has one interface specification record. The instance variables **text-update-form** and **text-analysis-form** specify the record.

Transformation Item Interface Instance Variables

The numbers and names of transformation item interface instance variables differ from one presentation type to another. The form of a transformation item interface instance variable usually includes one or more *probe value forms* which denote pieces of data from probes. A probe value form is a list consisting of a *probe key*, an *attribute key*, and an optional *save function*, and arguments if any for the save function if any save function is supplied. The probe key identifies a sender probe which is presumably defined in the **probes** instance variable of the panel. The attribute key is used to extract the value of a specific attribute of the data as the probe may send data which consist of more than one attribute value. The save function determines how to save the extracted attribute value into a states array and a display array. See section 6.2.7 for more details of save functions.

The following is an example of the definition of panel, **evaluator-queue-history-panel**. (See section 6.2.5 for the definition of **defpanel**). The **probes** interface instance variable for this panel has just one element implying that only one kind of probes are to be attached to this panel, which is the **evaluator-queue-probe**. (See secition 6.1.2 for its function and section 6.1.7 for its definition.) The probe key, **:queue-probe**, is defined in the **probes** instance variable initialization form and is referred to in the **left-axis-form** instance variable. The **left-axis-form** is one of the transformation item interface instance variables for a point and line presentation panel. The value denoted by the *probe value form*, **(:queue-probe :busy save-sum)**, would be a value of the **:busy** attribute of data received from a probe of **evaluator-queue-probe** type, identified by **:queue-probe**, and the value is saved into the corresponding element of a states array and a display array by the **save-sum** function (The **save-sum** is explained in the section 6.2.7).

A probe value form, considered as a whole, which appears in a form of a transformation item interface instance variable may be treated as an ordinary lisp expression, and therefore, can be enclosed by any kind of lisp expressions. This leads to the potential for specializing your own instrument (see section 6.2.7 for an example).

The following list summarizes the transformation item interface instance variables for available presentation types.

```
(defpanel evaluator-queue-history-panel
  ((tv:name              "EVALUATOR QUEUE HISTORY")
   (probes               '((:queue-probe evaluator-queue-probe)))
   (legend               " Total Evaluator Queue Lengths")
   (left-axis-form       '(:queue-probe :busy save-sum)))
  ((left-axis            (make-axis      :label  "Evaluator Queue Sum"
                                         :range  (make-range 0.0 nil))))
  (scrolling-line-panel-mixin))
```

- The **bal-presentation** has three transformation item interface instance variables; **line-points-form**, **line-density-form**, and **box-density-form**. The first two determines the attributes of lines drawn while the last determines the attribute of boxes drawn.
 - The **line-points-form** determines how to obtains a graphical point list to be drawn. The **net-output-connection-probe**, the **bus-output-connection-probe** and the **bus-input-connection-probe** provide as their **:points** attribute the type of a point list which a **bal-presentation** panel expects.[18]
 - The **line-density-form** determines how to obtain the thickness of a line to be drawn. The thickness is a positive integer where 1 indicates the density of an ordinary line and the bigger the number is, the thicker the density becomes.
 - The **box-density-form** determines the color or shade drawn inside a box. The value denoted by this form is in absolute application units. The actual color or shade for a given value is calculated according to the value of the **bal-presentation** instance variable **box-value-range** which indicates the range of possible values, and different colors or shades available.[19]
- The transformation item interface instance variables used for the histogram presentation types are **histogram-form**, **simulator-time-form** and **lower-histogram-form** where the **simulator-time-form** is only applicable to a **time-(dual-)histogram-plot-presentation** panel, and the **lower-histogram-form** is only applicable to a **(time-)dual-histogram-plot-presentation** panel.
 - The **histogram-form** determines how to calculate the height of each histogram bin for **(time-)histogram-plot-presentation** panels or for upper histograms of **(time-)dual-histogram-plot-presentation** panels. Values are in application units and the possible value range for the height is determined by the instance variable **left-axis**.

 The **lower-histogram-form** determines how to calculate the height of each lower histogram bin for a histogram panel with dual histograms. The value range of the lower histogram of a dual panel may be specialized by supplying an appropriate axis structure for the instance variable **lower-left-axis**

[18] The point list which a pal-presentation panel has extracted from probe data by :points attribute is supposed to be in the coordinates system where the design was created. The panel uses the slot s:currentTransform and SIMULA provided function **s:PointList** to transform it to the panel screen coordinates.

[19] An instrument window keeps available colors or shades in its instance variable s:box-alu-table for the panels of the instrument to use. By default an instrument has ten different colors or shades. If you want more or less different colors or shades for your instrument, make your instrument with init keyword :gray-levels set to the number you want. Which color or shade to use for each level is defined in the function make-box-alu-table and you can modify this function to suit your hardware or your preference.

- The **simulator-time-form** determines how to extract the time value from probe data for a **time(-dual)-histogram-plot-presentation** panel.

- There are two transformation item interface instance variables used by the **scrolling-bar-plot-presentation**.
 - The **bottom-axis-form** is usually a form to obtain the value of time from a probe.
 - The **left-axis-form** is a form to obtain a list of objects whose states should be displayed in bars.

- A panel of the point and line presentation type has three transformation item interface instance variables to determine its interface behavior.
 - The **bottom-axis-form** is a form to obtain an x coordinate of a point or a curve. The value range of the x axis may be adjusted by specializing the instance variable **bottom-axis**.
 - The **left-axis-form** is a form to obtain a y coordinate for a point or a curve which uses the left side y axis. The value range of the left side y axis is determined by the instance variable **left-axis**.
 - The **right-axis-form** is a form to obtain a y coordinate for a point or a curve which used the right side y axis. The value range of the right side y axis is determined by the instance variable **right axis**.

- The interface of the **scrolling-text-presentation** type is determined by one transformation item interface instance variable, **text-items-form**.
 - The **text-items-form** is a form to obtain a list of items to be displayed in a line.

6.2.5 Defining and Specializing Panels: Defpanel

This section describes the interface provided by SIMPLE to define new panels, and discusses the essentials of what this requires.

> **defpanel** *name init-instance-variables reset-instance-variables component-panels* **&rest** *options*

The syntax of the **defpanel** is similar to that of the **defflavor** except that it has two different kinds of instance variables; **init-instance-variables** and **reset-instance-variables**. A **defpanel** definition creates a flavor called *name* and the flavor inherits from all the flavors listed in **component-panels**. The **options** are handled in the same way as they are in any **defflavor** definition. The instance variables of the panel flavor include both **init-instance-variables** and **reset-instance-variables**. The major difference of an instance variable declared in a **defpanel** definition from an instance variable declared in a **defflavor** definition is that any initialization from of an instance variable can refer to any instance variable which is declared before it. In other words, the initializations of the instance variable values are performed in the strict order in which the instance variables are listed in the definition. Furthermore, the initialization of the value of an instance variable in **init-instance-variables** always precedes the initialization of any instance variable in **reset-instance-variables**.

An instance variable in **init-instance-variables** is such an instance variable that is going to have its value computed and set according to the value initialization form at make-instance time and every time a **:set-up** message[20] is received by the panel. An instance variable in **reset-instance-variables** is such an instance variable that is going to have its value computed and set according to the value initialization form not only at the instantiation time and the set-up time but also every time a **:reset** message[21] is received by the panel.

The *components-panels* argument may be a list of any panels or panel mixins which is defined by **defpanel** or any presentation type flavor such as **bal-presentation** or **line-plot-presentation**.

The following code is an example of **defpanel** definitions. It defines the panel mixin **site-mapping-panel-mixin** which has been used to define all the existing panels in the *box and line presentation* group (see the table in section 6.2.2).

Note that the initialization form of the instance variable **line-density-form** refers to another instance variable **line-density-function** which is declared above it and the initialization form of the instance variable **box-density-form** refers to another instance variable **calculator**. This panel has no **reset-instance-variables**. It is built on top of the **bal-presentation** flavor.

Specializing Probes

There are two useful instance variable of a probe to be used to specialize a probe in the *probes* instance variable form of a panel; **filter** and **selector**.

The value of **filter** should be a function which ,being executed in a **:calculate** method of the probe, controls the flow of data from a probe to a panel by applying the function to the arguments which the

[20] A **:set-up** message is sent to a panel when a panel is instantiated and when a new design is attached to the instrument to which the panel belongs.

[21] A .reset message can be sent to a panel whenever a user wants to, but it is usually called at the beginning of each simulation run through the function simple which is a SIMPLE facility to invoke a SIMPLE/CARE simulation.

```
(defpanel site-mapping-panel-mixin
  ((calculator)                ; presumably set by a superior panel
   (line-density-function      #'line-density)
   (selection-predicate        #'site-p)
   (sample-label-format        "~2D%")
   (sample-label-function
     #'(lambda (level levels) (* level (floor 100 levels))))
   (box-value-range            (make-range 0.0 1.0))
   (line-points-form           '(:connection-probe          :points))
   (line-density-form
     '(funcall ,line-density-function (:connection-probe :status)))
   (box-density-form
     '(funcall ,calculator (:queue-probe :busy))))
  ()                            ; no reset-instance-variables
  (bal-presentation))           ; component presentation
```

:calculate method takes.[22] If the function application returns **nil**, the data from the probe will not be forwarded to a panel.

The value of **selector** should be a function which, taking one argument, returns **T** if the argument is an object to connect a probe of this type to. A good example of it is shown below as a comparison between the definition of **evaluator-qload-scrolling-bar-panel** and that of **processor-qload-scrolling-bar-panel**. The latter uses the same probe, **evaluator-queue-probe**, as the former uses except that the probe is specialized with the **:selector** keyword so that the probe is only connected to a processor object with which the selector function **processor-selector** returns **T**.

[22] Those are probed-object, probe-key, update-items and probe-time.

```
(defpanel evaluator-qload-scrolling-bar-panel
  ((tv:name              "EVALUATOR QUEUE LOAD")
   (probes               '((:queue-probe evaluator-queue-probe)))
   (legend               "Recent History and Average by Site")
   (object-list-function
     #'(lambda (box) (collect-objects box #'site-p))))
  ()
  (scrolling-bar-panel-mixin))


(defpanel processor-qload-scrolling-bar-panel
  ((tv:name              "PROCESSOR QUEUE LOAD")
   (probes               '((:queue-probe evaluator-queue-probe
                            :selector processor-selector)))
   (object-list-function        'all-processors))
  ()
  (scrolling-bar-panel-mixin)
  (:Documentation
    "Like a normal evaluator-qload-scrolling-bar-plot-panel only it
 connects itself only to Processor type sites for the shared memory
 model."))
```

6.2.6 Panel :Update Method

A panel receives from a probe data in the message **:update** after a sequence of messages **:trigger**, **:calculate** and **:select** are handled by the probe. An **:update** message consists of four things; a *probe key* keyword which allows a panel to identify the sender probe of the message, a *probed object* (often the probed component's enclosing site, in CARE), *update items* (a disembodied property list containing the abstracted state data on this object), and *probe time* which is the simulated time at which the data was collected. In order to simplify the process of designing a new panel, the above four values are stored in global variables as soon as a panel receives an :update message from a probe. Those are called ***probe-key***, ***probed-object***, ***update-items***, and ***probe-time*** respectively. These global variables may be used to customize the *update*, *save* or *analyze* behavior of your panel. There is another global variable, *spec*, which indicates the panel interface specification object with which the panel is processing the :update message.

```
(defmethod (basic-presentation :update)
           (probed-object probe-key update-items &optional probe-time)
  (setq *probe-object* probed-object *probe-key* probe-key
        *update-items* update-items *probe-time* probe-time)
  (loop for spec in interface-specifications do
        (setq *spec* spec)
        (funcall (panel-update-closure spec))))
```

A user who intends to customize interface behaviors may use the SIMPLE provided construct, **with-update-spec-bindings**, to fetch those globals into your local environment. The following piece of code is an example of **with-update-spec-bindings** usage. It defines the default behavior of the update method for an interface specification object of a panel.

```
(defmethod (basic-presentation :update-states-array) ()
  (with-update-spec-bindings
    (spec save-function probed-object update-items)
    (with-object-states-bindings
      (old-states old-display new-states new-display)
      spec probed-object
      (funcall save-function new-states update-items new-display)
      (unless old-display
        (enqueue new-display (panel-states-arrays-updated spec))))))
```

The local variable, **spec**, **save-function**, **probed-object** and **update-items**, set by the **with-update-spec-bindings**, contain the current interface specification record, the save function for the record, the probed object, and the abstracted state data of the object respectively. The SIMPLE provided construct, **with-object-states-bindings**, fetches appropriate values for **old-states**, **old-display**, **new-states** and **new-display** by hashing with **probed-object** as a key **states-arrays-hash** and **states-arrays-display-hash** of the spec. The **old-states** is a states array which contains

the last states of the **probed-object**. The **old-display** is an display array of the **probed-object** which was used for the last time the panel refreshed its contents. The **new-states** is a states array which will contain new states of the object. The **new-display** is a display array which will be used for the next display. Then, the **save-function** of the spec record is applied to update specific elements of the **new-states** and the **new-display**. Finally the **new-display** is queued to the **panel-states-arrays-updated** queue of the **spec** to be displayed (see section 6.2.3 for the structure of the interface specification record and the explanation of each element).

6.2.7 Customizing Interface Behaviors

The facility of customizing interface is divided into three parts for the modularity and for the ease of interface customization.

Customizing Update Forms

The behavior of the **:update** method of a panel is determined by the **update-closure** element of each interface specification. Customizing the behavior of **update-closure** of an interface specification record can be done by specializing a corresponding update form interface instance variable (see section 6.2.4 for the explanation of various interface instance variables). If the value of an update form is left `nil`, then a default form is filled in.[23] One easy way to customize an update form is to define a method, say *:my-update*, for an appropriate presentation type and provide '(send self :my-update *args*) for the value of the update form instance variable. Some of the examples of such methods defined for existing panel types are:

- **:update-time** method defined for scrolling plot presentations. It takes one argument to indicate the element of an array which keeps the simulated time.
- **:update-interval** method defined for the **time(-dual)-histogram-plot-presentation**. It also takes one argument to indicate the element of an array which keeps the interval of time that the object has spent in the state represented by the array.
- **:update-class** methods defined for the **scrolling-text-presentation** and used for the **class-activity-panel**. It takes three arguments; the first is the attribute name which is used to get the class of a probed object with, the second indicates an element in an array where the name of the class is kept, and the third indicates an element in an array where the total number of instances of the class is kept.

Customizing Analysis Forms

The method of customizing the **analysis-closure** of an interface specification record is similar to that of customizing the **update-closure**. That is, to specialize a corresponding analysis form instance variable. All the examples of analysis forms found in existing panels are using the function **sort-arrays** which sorts collected display arrays before they are displayed.

sort-arrays *conditions* **&optional** *target*

- The *conditions* argument determines how to sort display arrays. It is a list of a *condition* which indicates one way of sorting, and therefore display arrays can be sorted by more than once. A *condition* consists of a sorting function, a probe value form denoting an element in an array by which display arrays are to be sorted, and any number of probe value forms which are optional and can be supplied if the sorting function needs values of array elements denoted by them to perform desired sorting.
- The *target* argument which is optional allows you to keep the sorting result in a specific element of each array. The argument is a probe value form denoting an element in an array.

[23] The form is '(send self :update-states-arrays) and a panel of any type has an :update-states-arrays method defined or inherited.

The following piece of code is used for the analysis form of the panel **cumulative-latency-panel** (see section 6.2.2 for its function). The conditions argument consists of one condition which sorts display arrays in decreasing order of the value of an array element which keeps the sum of three latency values extracted from the data received from a probe represented by **:latency-probe** key. After the sorting is completed, the rank number of each display array which indicates a position of the array in the sorted list is stored in an element of the array denoted by the form **(:cumulative-latency :rank)**.[24]

```
(sort-arrays
  (list (list #'>
              (:latency-probe
                (+ :launch-delay :net-delay :operator-delay :evaluator-delay))))
  (:cumulative-latency :rank))
```

Customizing Save Functions

As explained in a section 6.2.4, a transformation item interface instance variable includes one or more probe value forms The first and second element of a probe value form are required and denotes a probe identifying keyword and an attribute keyword respectively. You can supply the third argument, *save function*, to indicate how to save an attribute value extracted from probe data into an array element. Therefore, the general syntax of a probe value form is a probe key, an attribute key, a save function, followed by any special arguments for the save function. Providing no save function has the same effect as providing the default function, **save-states**. The **save-states** takes four arguments. Any save function which needs other arguments than the arguments defined for the **save-states** function has to list the additional arguments in its probe value form.

save-states *new-value slot current-states display-states*

The argument **new-value** contains an extracted attribute value of an probed object, the argument **slot** indicates the position in an array where **new-value**, processed if necessary, should be saved, the **current-states** indicates a states array for the object, and the **display-states** indicates a display array of the object. The **save-states** function saves the value of **new-value** into the **slot** position of the **current-states** and that of the **display-states**.

Aside from the **save-states**, there are various save functions defined and available.

save-summation is a function which saves **new-value** into **current-states** as **save-states** does. The value saved into **display-states**, however, is calculated by adding **new-value** to the result of subtracting the "old value" from the current display value, where the "old value" may be found in the **slot** element of **current-states** and the current display value may be found in the **slot** element of **display-states**. The **slot** element of **display-states** is thus used to keep the sum of **slot** element values of all the states arrays

accumulate is a function which does not use **current-states** and just increments the **slot** element value of **display-states** by **new-value**

[24] The form (:cumulative-latency :rank) has a similar syntax as a probe value form and it does indicate an element in a display array. The :cumulative-latency, however, does not represent any probe and the form does not denote a value from a probe as a probe value form does. Instead, this form simply reserves an element in a display array to keep its rank number.

save-accumulation : is a function which takes two special arguments, **accumulation-slot**, and **count-slot**. Its function is similar to the function of the **save-sum** except that it keeps track of not only the sum of values, but the count which denotes how many times the function is executed for the object which the **current-states** represents. The sum is kept in the **accumulation-slot** element of a display array and the count is kept in the **count-slot** element of a display array.

save-modified-sum : is not exactly a save function which can appear in a probe value form. Instead, it creates a save function. A save function created by the **save-modified-sum** with one function argument called **translation-function** is a function which works in a similar manner to the **save-sum**. The difference is that the **translation-function** is applied to both the old value of the slot and the new value before calculating the value to be saved into **display-states**.

save-count : is also an example of a function which creates a save function, as the **save-modified-sum** does. A save function that is created by the **save-count** with two function arguments, **interval-function** and **initializer**, keeps track of the number of occurrences of events which are grouped into several intervals by **interval-function**. The **initializer** is used to initialize the **slot** element of **display-states** to an appropriate structure.

Customizing Extra Manipulation over Probe Data

Data from a probe, being saved into a corresponding element of a states array and a display array according to the save function, may need to be further processed to assure proper interface between the panel and the probe, or to meet the goal pertinent to your design. As explained in a section 6.2.4, a transformation item interface instance variable may have any complex lisp expressions surrounding a probe value form to suit your instrument design. For example, if you want a transformation item interface instance variable to have the average value of data from two probes, then the proper form would be:

```
(/ (+ (:probe-1 :attribute-1) (:probe-2 :attribute-2)) 2)
```

where the probe value forms, *(:probe-1 :attribute-1)* and *(:probe-2 :attribute-2)*, denote data from two probes.

6.3 Instruments

This section describes the instruments available in CARE and the SIMPLE interface for users to define new instruments or specializing existing panels.

An instrument is a collection of panels each of which is associated with a specific set of probes. Hence, the definition of an instrument is simple once you define by **defpanel** the component panels with appropriate probes which may in turn be defined by **defprobe**.

6.3.1 Defining and Specializing Instruments: Definstrument

A new instrument can be defined by the SIMPLE construct **definstrument**.

> **definstrument** *name* **&key** *print-name components panels configurations selected-panel selected-package init-ivs documentation*

It creates a flavor named *name* and eight keyword arguments are handled as follows.

- **Print-name** is the name of the instrument window. It may be used in an underlying lisp machine window system to identify the window.
- **Components** is a list of component flavors to be mixed into this instrument frame.
- **Panels** is a list of any of the following two forms:
 - an atom which which is a name of a panel type defined by defpanel.
 - a list of a panel name, a panel type, and alternating keyword instance variable names and their values. The panel name is a name for the panel used in the instrument to identify it and the panel type is a name of a panel type defined by defpanel. Suppose *Panel-A* is a defined panel, then *(my-panel Panel-A :iv1 iv1-value :iv2 iv2-value)* is a valid component of **Panels** if iv1 and iv2 are instance variables of Panel-A. My-panel only differs from the default Panel-A in terms of the values of the instance variables.
- **Configurations** specifies the constraints for the instrument window. This can be any of the following three forms:
 - a constraint definition like that you would give to an ordinary constraint frame definition, eg. ((config1 . ((.....))))
 - an expression whose value will be a list of constraints. This could well be a call to any of the constraint frame cliche functions.
 - Nil. In this case the system will generate a simple constraint frame for you.
- **Selected-panel** is the name of the panel which will be the selected pane of the instrument window.
- **Selected-package** is a package in which the user environment of the instrument resides.
- **Init-ivs** is a list of (name value) pairs, which provide values for some instance variables of the instrument constraint frame defined.
- **Documentation** is a documentation string for the instrument.

6.3.2 Available Configurations

The following is a list of available functions to create configurations.[25]

- **simple-multiple-configurations** takes one argument, **panes** (a list), and creates N different configurations where N is the number of panes in **panes**. Each configuration is named 0, 1, ..., N-1, N and the configuration named *n* contains n number of panes which are the last n panes listed in **panes**.
- **big-box-in-the-middle** takes five arguments, **big-panel**, **left-group**, **middle-group**, **right-group** and **documentation-panel**, and creates a constraint frame with a big panel for **big-panel** in the middle at the top. There is **documentation-panel** at the bottom. The other panes are split into three groups; those which go down the left hand side of the screen, i.e. **left-group**, those which are at the bottom, i.e. **middle-group**, and those which are on the right, i.e. **right-group**.
- **big-box-in-the-middle-with-swapable-top-and-bottom-left-panels** is just like the **big-box-in-the-middle** except that an extra set of configurations are provided, which swap the top left panel or the bottom left panel for the big box panel.
- **two-rows-with-documentation** takes three arguments, **top-group**, **bottom-group** and **documentation-panel**. **Documentation-panel** lies at the bottom and the rest of the screen is divided horizontally into two halves, whose top half contains top-group panels of equal size, and whose bottom half contains bottom-group panels of equal size.

[25] The names of all the available functions can be found in the variable s:*all-constraint-frame-cliches*.

6.3.3 Mouse Changeable Attributes

By clicking a middle button on any region of an instrument window, you can bring up a menu consisting of various actions which can be taken for the instrument and the simulator to which the instrument is attached. Menu items currently available are:

Activate/Deactivate Panels : It brings up another menu which consists of a list of all panel names which are associated with probes. By highlighting or unhighlighting them, you can activate or deactivate the display actions of panels respectively.

Activate/Deactivate Probes : It brings up another menu which consists of a list of all probe names used in the instrument. By highlighting or unhighlighting them, you can activate or deactivate the monitoring actions of probes respectively.

Modify Instrument Attributes : Various instrument attributes can be modified through a menu.[26]

Modify Simulation Parameters : Various simulation parameters, such as processor performance of the design which is being simulated, can be modified through a menu.

Set Configuration : The configuration of the current instrument window can be changed to one of the configuration defined for the window.

Change Screen : The screen on which the current instrument window is created can be swapped between **black-and-white** and **color** if a color monitor is available.

Hardcopy : It prints a hardcopy of the current instrument window image.[27]

Inspect : It puts you into an inspector window which will inspect the current instrument window object.

[26] Currently two attributes are in the menu and those are related with an instrument which manpulates time weighted averages of values.

[27] You can specify a printer by setting s'*simulator-image-printer*.

6.3.4 Existing Instruments

There are several instruments defined by **definstrument**.[28]

- **observer** has ten panels with configurations created by the function **big-box-in-the-middle-with-swapable-top-and-bottom-left-panels**. The components panels are **net-operator-qload-site-Mapping-Panel**, **evaluator-operator-histogram-panel**, **cumulative-Latency-Panel**, **network-offered-load-latency-Panel**, **operator-potential-latency-panel**, **evaluator-potential-Latency-Panel**, **instance-Activity-Panel**, **class-Activity-Panel**, **lisp-Panel**, and **notes-panel** with the **lisp-panel** in the biggest pane.
- **examiner** is a similar instrument to **observer**. The biggest pane is used by **evaluator-qload-scrolling-bar-panel** and two potential latency panels, **evaluator-potential-latency-panel** and **operator-potential-latency-panel**, are missing.
- **basic-shared** is an instrument for a shared memory design. The components panels are **processor-qload-scrolling-bar-panel**, **memory-qload-scrolling-bar-panel**, **processor-memory-histogram-panel**, **network-latency-panel**, **processor-queue-history-panel**, **lisp-panel**, **notes-panel**, and **processor-and-memory-qload-mapping-panel**.
- **bus-observer** is an instrument for a bus design. It is just like the **observer** except that it uses a special mapping panel, **bus-operator-qload-mapping-panel**, to show the activities around buses.
- **sv-bus-observer** is an instrument which is just like **basic-shared** except for the use of a special mapping panel for a bus design.
- **bus-examiner** is an instrument which is just like the **examiner** except that it uses a special mapping panel to show the activities around buses.

[28] The names of all defined instruments can be found in the global variable s:*all-instrument-names*.

Acknowledgements

Numerous people have made significant contributions to SIMPLE/CARE. They include Harold Brown, Gordon Foyster, Max Hailperin, Russ Nakano, James Rice, Eric Schoen, Manu Thapar, Tony Waitz, and Jerry Yan.

We acknowledge the patience and invaluable feedback of the users of SIMPLE/CAREwho have helped direct the development of the system. We would also like to thank the staff of the Symbolic Systems Resources Group of the Knowledge Systems Laboratory for their excellent support of our computing environment. Special thanks to Ed Feigenbaum for his continued leadership and support of the Knowledge Systems Laboratory and the Advanced Architectures Project, which made the development of SIMPLE/CARE possible. This work was supported by DARPA Contract F30602-85-C0012, by NASA Ames Contract NCC 2-220-S1, by Boeing Contract W266875, and by Digital Equipment Corporation.

Bibliography

[1] Harold Brown, Christopher Tong, and Gordon Foyster. Palladio: An exploratory environment for circuit design. *Computer*, 16(12):41–58, December 1983.

[2] Max Hailperin. Private communication.

[3] Guy L. Steele Jr *Common Lisp: The Language*. Digital Press, 1984.

Appendix A

Installing SIMPLE/CARE

Release 0 of SIMPLE/CARE is available both via cartridge tape and via anonymous FTP from *sumex-aim.stanford.edu.*

This section describes the software provided on the tape, and the mechanics of installing the system at a remote site.

A.1 Systems

A **system** is a collection of logically related files that are managed together. The Lisp Machine system facilities allow a user to specify a group of files to be loaded, as well as the compilation and load-time dependencies among them. (This is similar to the *make* facility on Unix(tm) systems.)

The SIMPLE-CARE system is simply a shell system that loads in the definitions of numerous other systems described below.

The CARE system contains all the files which define the SIMPLE/CARE simulator. It is composed of the following four subsystems:

1. BAL: Implements the low-level component structure and behavior representations and the event driven simulator of SIMPLE.
2. SIMPLE: Implements the basic instrumentation of SIMPLE.
3. CARE-COMPONENTS: Defines the behavior of the simulated components used to build multiprocessor architectures. Also defines the CARE applications interface (*Lamina*).
4. CARE-INSTRUMENTS: Defines the instruments, panelspecs and probes used to monitor performance of CARE designs.

LAMINA application systems called LINESIM and ELINT are also supplied with the release.

The graphical editor for constructing designs and components is defined by the P-HELIOS system. This is not required to be loaded when executing simulations.

A.2 Installation on *TI Explorer* Machines

Release 9 of SIMPLE/CARE is currently supported for Release 6 on *TI Explorer* machines.

1. Restore the directory structure obtained via tape or FTP. You should get a top-level directory corresponding to *HOST:SIMPLE-CARE.V0;*.
2. Edit the file *HOST:SIMPLE-CARE.V0;CONFIG.LISP* and change the references to the host **AAP** to your host machine corresponding to *HOST* in the pathname. Write out the changed file.
3. Edit the file *HOST:SIMPLE-CARE.V0;SIMPLE-CARE.SYSTEM* and change the reference to the host **AAP** to your host machine corresponding to *HOST* in the pathname. Write out the changed file as *SYS:SITE;SIMPLE-CARE.SYSTEM.*
4. Do **(make-system 'simple-care :noconfirm)** to load the definitions of the systems that make up the SIMPLE/CARE system.
5. Now you can do **(make-system 'care :noconfirm)** to load up CARE.

Appendix B

A Dynamic, Cut-Through Communications Protocol with Multicast

Also available as:

Technical Report No. KSL-87-44
Knowledge Systems Laboratory
Department of Computer Science
Stanford University
Stanford, CA 94305

Appendix C

HELIOS User's Manual

Also available as:

Technical Report No. HPP-84-34
Knowledge Systems Laboratory
Department of Computer Science
Stanford University
Stanford, CA 94305

Knowledge Systems Laboratory
Report No. KSL-88-33

July 1988

ELINT in LAMINA
Application of a Concurrent Object Language
(Extended Abstract)

Bruce A. Delagi and Nakul P. Saraiya

KNOWLEDGE SYSTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, CA 94305

Presented to:
Workshop on Object-Based Concurrent Programming OOPSLA 88

ELINT in LAMINA

Application of a Concurrent Object Language

(Extended Abstract) *

Bruce A. Delagi Nakul P. Saraiya

July 27, 1988

Abstract

The design and performance of an "expert system" signal interpretion application written in a concurrent object-based programming language, LAMINA, is described together with a synopsis of the programming model that forms the foundation of the language. The effects of load balancing and the limits imposed by task granularity and message transmission costs are studied and their consequences to application performance are measured over the range of one to 250 processors as simulated in SIMPLE/CARE, an extensively instrumented simulation system and computer array model.

*This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266675.

1 Foundations of the LAMINA Object Model

The LAMINA object programming model is based on asynchronous communicating objects. The objects communicate using streams. An *object*, as used here, is a collection of variables, the *state variables* of that object, manipulated by (and only by) a set of procedures, the *methods* associated with that object. Streams represent sequences of values over time; information sent to a stream builds the sequence represented by that stream.

Each LAMINA object has associated with it a distinguished stream that is its *task stream*. The information arriving on an object's task stream specifies tasks for the object; each such piece of information is a *message*. Each message names a method to execute and includes the parameters for the execution. When a task execution sends a message to a stream, the execution is not normally delayed to wait for a responding message (or even for an acknowlegement of the receipt of the message).

The information sent to a stream consists of references to streams, and unshared values, which may be both atoms and structures. Values that have internal structure must be *encoded* before transmission. Encoding involves both graph structure linearization and internal pointer relativization. When such a value arrives at its destination, storage must be allocated to contain it and it must be *decoded*, that is, internal pointers must be reexpressed in absolute terms.

Like ACTORS[1], the LAMINA object model is characterized by non-deterministic receipt of messages; message arrival order is not guaranteed to be in sending order. Like ACTORS, message arrival triggers computation. In other ways, however, as discussed in the full paper, the LAMINA object model departs from ACTORS, by generally trading off flexibility for efficiency, by dealing

more directly with mutability, and, since streams are first-class entities, by allowing objects to establish communications over streams other than their task streams.

1.1 Computational Flow

As illustrated in figure 1, messages arriving on the task stream of an object specify tasks to be done by that object. There is an eternal dispatch process for each object which takes these messages from the stream and executes them in turn.

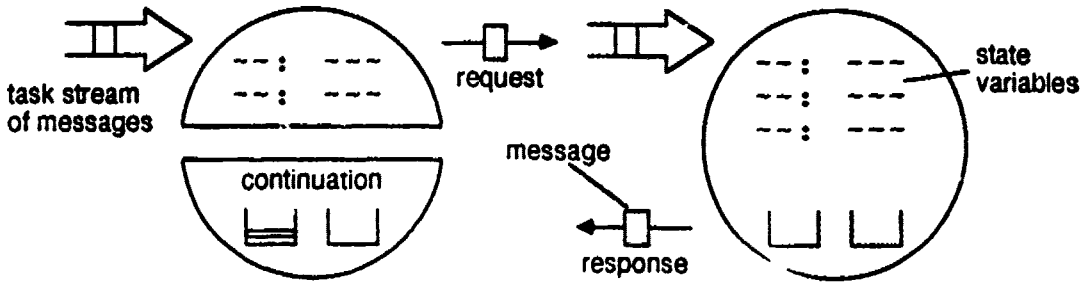


Figure 1: Message Passing Model with Continuations

Tasks usually mutate the state variables of the object and generate new messages. Tasks have exclusive access to their execution context but are preemptible and can also have implicit continuations.

Tasks in the LAMINA object model are normally *data driven* and *run to completion.* They are generally intended to be accomplished as the stages of a pipeline, thus organizing the work performed by the objects of the application. Objects only begin tasks when all the needed information

is available. In order not to block the pipeline, a task that is started is run to completion unless it is preempted by the underlying system (*e.g.* for a debug trap or the consumption of run quanta).

Experience with the LAMINA object model[3,4] has demonstrated that, with few exceptions, the continuation of a task is most readily specified explicitly as a message that is sent to an object. When this is not sufficient, an implicit, anonymous continuation (as shown in figure 1) is used to capture the environment needed to later continue the computation, and this is deferred until further information is available; the object may perform other tasks while awaiting the required information. To form the continuation, any required bindings that are on the stack are copied into a closure and the stack storage is released. Stack allocation is thus used to the greatest extent possible and heap allocation is minimized.

1.2 Reusing Physical Stack Storage Space

The internal state of a LAMINA object (its state variables) is expected to occupy on the order of tens to hundreds of bytes. Method execution is expected to require on the order of hundreds to thousands of instructions.

The binding and control stacks for both a task and its (implicit) continuation are empty when execution is begun, non-empty during execution, and empty again when execution is done. Since task preemption is an exceptional condition and since tasks and their continuations otherwise always run to completion, stack storage space is generally reuseable among all the tasks on a processor. This avoids the high space penalty of using coarse-grained page-protection-based stack limit mechanisms, allowing the use of efficient virtual memory and cache mechanisms without resorting to coarse-grained task decomposition.

1.3 The Illusion of Atomicity

When the system preempts an object's task, that object does not execute any other tasks until the preemption is resolved. In this way, while the object's pipeline is indeed blocked while the preemption exception is dealt with, the illusion of atomic execution with respect to the context of a task is preserved. However, tasks for other objects may be run as long as their local execution conditions are satisfied. This means that data consistency can only be preserved if no state is shared between objects. LAMINA objects never share structures; they communicate only by exchanging messages, which may contain *independent* copies of local structures. Thus the atomicity of operations on an object is not affected by the operations on other objects.

Implicit continuations are not part of the original task's atomic execution. Instead, the task and its continuation are independent atomic executions. The execution of the original task is first completed and its continuation is executed some time later, when the latter's requirements for additional information have been satisfied. In the meantime, other tasks are executed by the object, allowing messages specifying additional work to be passed down the pipeline to other objects.

Although an implicit continuation is a separate atomic execution, it shares the spawning object's execution environment. Therefore, any structures which are closed over may be altered by other tasks on the same object while the continuation awaits execution: invariants must be reestablished by the completion of each task and continuation.

2 Design of the ELINT Application

ELINT[5] is a real-time system for interpreting processed, passively acquired radar emissions from

aircraft in a monitored airspace. It correlates the radar emissions that are observed by multiple, mobile detection sites into the individual radar emitters producing those emissions. It then fuses these emitters into clusters of emitters that are co-located over time. It also maintains the track and activity histories of the clusters, and hypothesizes the types and number of aircraft in the clusters based on their constituent emitters.

2.1 Pipeline Organization

ELINT is naturally organized as parallel, data-driven object pipelines, as shown in figure 2.

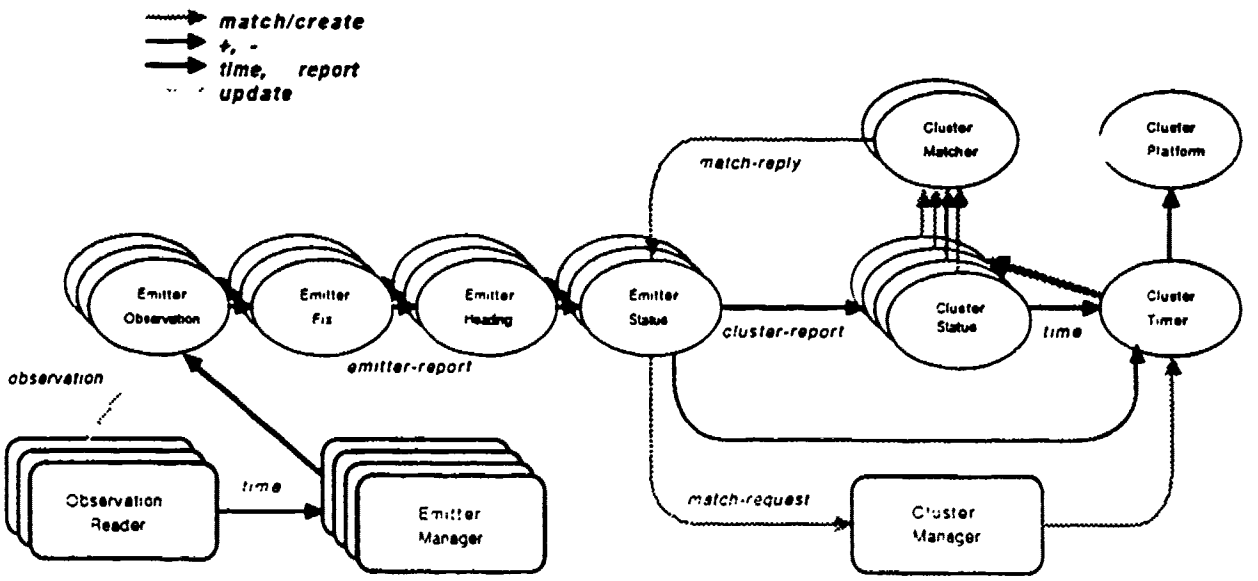


Figure 2: ELINT Pipeline Organization

Observation readers read in time tagged **observation** structures, representing observed emissions, and pass these on to the **emitter observations** for the identified emitters. Here they are "buffered" until the end of the current data timeslice is detected, when they are abstracted into an **emitter-report** structure and started down the pipeline. Successive pipeline stages compute and

attach track and status information to this report.

Emitter status object link their respective emitters to an appropriate cluster, and, once clustered, propagate their latest status information on to update the cluster. Each group of **cluster status** objects implements a distributed database of the track and activity history of a cluster; this is used both to report on the cluster periodically and to match against the tracks of new emitters that attempt to cluster.

2.2 Replication

Replication is a useful means of relieving congestion. It is viable to the extent the replicated objects have no dependencies between them. Early experience indicated that the **observation readers** were obvious candidates for replication, since the rest of the system was often data starved. **Emitter managers** are also replicated to scale with the size of the system: a simple modulo operation on an emitter's identifier is used to break dependencies while maintaining consistency.

Sometimes the benefits of replication sufficiently outweigh the need to maintain consistency at all times, especially if the system has the capacity to detect and later correct the inconsistency as part of its regular problem solving activites. This is the case with the **cluster status** objects, which form the database of a cluster's track and activity history. The database is partitioned based on data time. As a consequence, **cluster status** objects detect "emitter splits" (i.e. emitters that were part of the cluster but whose tracks have now diverged from that of the cluster) in isolation: this can lead to inconsistent decisions as to the particular emitter that is split off, as a result of different message arrival orders at the objects (since the first arrival determines the inherited track of the cluster for that data time). However, at worst, too many emitters are split off, and the

system recovers because these emitters retry clustering with all extant clusters. The benefits of replication here are two-fold: besides reducing congestion at the cluster, the grain size of the match performed at each object during clustering is also reduced.

2.3 Clustering: Trees in a Distributed Loop

An emitter that has been detected for a sufficient period of time must either link to an existing cluster or create a new one. This computation is organized as a distributed loop (see figure 2). The **emitter status** object creates a **match-request** structure containing the track history of the emitter and a counter of the number of clusters that it has already (unsuccessfully) matched against, and it sends this to the **cluster manager**. The latter maintains a local list of all extant clusters; it multicasts the **match-request** to the **cluster timers** of those clusters that have not been matched against, after incrementing the counter to the total number of clusters. If the counter shows that matches against all extant clusters have failed, the loop terminates and the **cluster manager** creates a new cluster for the emitter.

A **cluster timer** defers the request until it is coincident in time, whereupon it multicasts the request to its **cluster status** objects, specifying a randomly selected **cluster matcher** as the intermediate client of the match. The **cluster matcher** collects the results of the partitioned match and then sends a **match-reply** structure to the original **emitter status** client, which awaits either a successful reply from one cluster or unsuccessful replies from all the clusters before determining whether to continue or terminate the loop.

In this way, an emitter can match against multiple clusters concurrently, a cluster can match against many emitters concurrently, and a cluster can match the blocks of its partitioned history

against an emitter concurrently.

2.4 Load Balance and Other Considerations

ELINT uses only static allocation of objects with multiple allocators—the managers. Random allocation performed well when the number of objects was far greater than the number of processing sites. Performance was improved by partitioning the sites based on object classes during initialization and allocating randomly within each block during runtime; the partition was based on the relative measured activity of the classes. This reduced both the load variance and the interference between pipeline stages.

Other notable features of this implementation are the distributed maintenance of data time and the enforcement of arrival ordering through the use of *sequenced streams*, which allow the consumer of the stream to access it only in accordance with the sequence numbers assigned to items put on the stream by its producers. Implicit continuations are used to hide the delay associated with creating new objects without blocking the task pipelines.

3 Performance of the Application

ELINT has been implemented on the SIMPLE/CARE[2] simulation system. The full paper details the components and their critical performance parameters; in brief, ELINT task granularities ranged from 80 to 300 microseconds and message sizes ranged from 10 to 50 words, with transmission latencies from 100 microseconds to a few milliseconds.

3.1 Making Measurements

ELINT was evaluated with respect to *correctness* and *timeliness*. Correctness was determined using a number of different *scenarios* (data sets) which exercised all the decision-making capabilities of the system. Timeliness was used to determine speedup. Since ELINT is a simulation of a real-time system, *sustainable data rate* was the speed metric rather than time to completion. For a given scenario and processor population, data was pushed into the system at the fastest possible rate such that the *latencies* of key inferences (*e.g.* cluster fixes) did not grow over time.

3.2 Results

ELINT was found to perform well with respect to correctness for a number of scenarios.

The timeliness results for a typical scenario are presented in figure 3, in which 20 emitters form 4 clusters over 50 data time units.

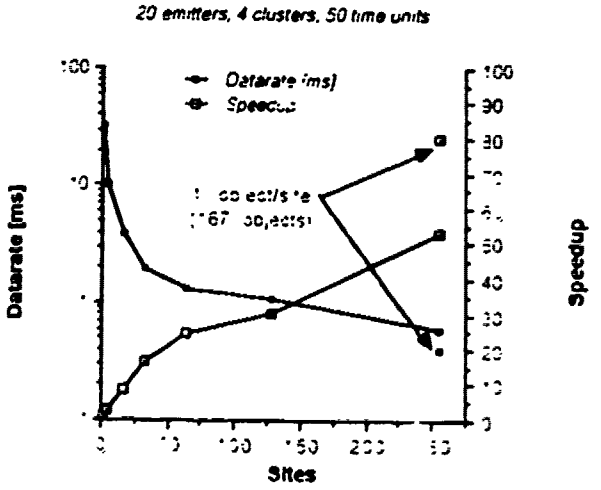


Figure 3: Data Rate and Speedup

3.3 Conclusions

With one object per site, ELINT can sustain a data rate of 400 microseconds, a speedup of 80 over the serial case. `Emitter` observations have a task granularity of 100 microseconds and they each receive and process three messages per cycle before initiating further pipeline activity. These times determine a "hard" limit on the performance of the system that is independent of the size of the data set; the limit is approached in this case.

As the load on the sites increases with smaller processing arrays, the contention for site resources degrades performance, since pipelines can only go as fast as their slowest stage. To a first approximation, the throughput of the system is limited by the throughput of the worst loaded site, and variance in the load across sites accounts for sub-linear performance.

Clustering and dynamic object creation latencies are fixed costs (on the order of milliseconds) that become more dominant with higher data rates, because work that is deferred until clustering or creation is accomplished "costs" more data time units. Other factors that affect performance are (1) the object pipelines are only approximately balanced, (2) contention for site resources (both message handling and application code processing) degrade pipeline balance through interference, (3) replication factors are computed without prior knowledge of the size of the data set, and (4) there are parts of the system that are not pipelined, but which feed back into the pipe.

3.3 Conclusions

We have described a concurrent object model and the implementation and simulated performance of ELINT, a signal understanding application, that uses this model. The design of ELINT was extensively refined as we gained better insights into its behavior through the instrumentation facilities provided by the SIMPLE/CARE computer simulation system. ELINT was best programmed using

explicit continuations; implicit continuations were not found to be important. Once the system had been tuned to balance pipelines, load imbalances reduced system efficiency, primarily by introducing pipeline imbalances due to resource allocation considerations. In the perfectly loaded case, ELINT was seen to achieve the hard performance limits set by application task granularity and message handling costs.

References

[1] Gul Agha. An overview of actor languages. *SIGPLAN Notices*, 21(10):58 – 67, October 1986.

[2] Bruce A. Delagi, Nakul P. Saraiya, Sayuri Nishimura, and Gregory T. Byrd. *An Instrumented Architectural Simulation System.* Technical Report KSL-86-36, Knowledge Systems Laboratory, Stanford University, 1986.

[3] Russell Nakano, Masafumi Minami, and John Delaney. Experiments with a knowledge-based system on a multiprocessor. In *Third International Conference on Supercomputing Proceedings*, 1988.

[4] Alan Noble and Chris Rogers. *AIRTRAC Path Association: Development of a Knowledge-Based System for a Multiprocessor.* Technical Report KSL-88-44, Knowledge Systems Laboratory, Stanford University, 1988.

[5] Mark Williams, Harold Brown, and Terry Barnes. *TRICERO Design Description.* Technical Report ESL-NS539, ESL, Inc., May 1984.

Knowledge Systems Laboratory
Report No. KSL-86-20

March 1986

Multi-System Report Integration Using Blackboards

by

J. R. Delaney

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

ACKNOWLEDGEMENT

This work was supported by the Defense Advanced Research Projects Agency, the NASA-Ames Research Center, Boeing Computer Services, and the National Institutes of Health.

ABSTRACT

Blackboards are an AI problem solving methodology. A blackboard system consists of a structured data base (the blackboard) holding input and derived inferences and a collection of procedures for deriving inferences (knowledge sources). Each knowledge source is specialized to operate on some portion of the blackboard. The knowledge sources are invoked opportunistically as the information on the blackboard increases.

The best known applications of the blackboard methodology have been in speech understanding and passive sonar data interpretation. The inputs in these cases were a single form of raw sensor data. But the methodology is also well suited to integrating multiple streams of fully reduced and qualitatively different data such as active radar track reports, passive electronic intelligence reports, and human intelligence reports about enemy intentions.

This paper sketches the nature of the blackboard problem solving methodology with an emphasis on those features suiting it to such applications. The sketch is illustrated with examples from a relatively simple multi-system report integration problem. Relevant applications currently under development at Stanford's Knowledge Systems Laboratory are also described.

1. INTRODUCTION

"Multi-System Report Integration" is an odd phrase. An alternative would have been "Sensor Data Fusion". But that phrase often implies a less reduced form of information to integrate than is intended here. The reporting systems in this paper are presumed to reduce the data they sense as fully as is practical with only that data available. The degree of processing can vary from system to system. For a radar tracking system, the reports would be samples of on-going tracks integrating all measurements up to the present. For an ELINT system dealing with intermittent emissions, the reports might be just current emitter and bearing characteristics. And for a human intelligence gathering system, the reports might be informed guesses about near-term enemy intentions.

"Sensor Data Fusion" also usually implies that the information to be integrated appears at comparable time intervals or is static. But the reporting systems in this paper are presumed to provide reduced data over a wide range of time intervals. The radar, ELINT, and "humint" systems mentioned above could produce reports at very different intervals with very different degrees of regularity. Assuming that some reports are locally of comparable frequency while others are locally static information is Procrustean.

"Blackboards" refers to a particular AI problem solving methodology. The best known applications of the blackboard methodology are HEARSAY-II, a speech understanding system (2), and the HASP/SIAP sonar data interpretation system (4,5). These applications effectively processed regular streams of data from a single sensor, treating any other information as locally static. But the blackboard methodology is more generally applicable. In particular, it provides a convenient framework for integrating maximally reduced information from multiple sources with different temporal characteristics. Just what is needed for multi-system report integration.

In the first section below, the fundamental features of blackboard systems are described abstractly. A consistent set of examples are used in the following section to clarify those features in context of multi-system report integration. The next section reviews those aspects of the blackboard methodology particularly suited to multi-system report integration. The last section briefly describes work in progress at Stanford's Knowledge System Laboratory on two more ambitious examples. It also explains how that work is embedded in a larger effort.

2. NATURE OF BLACKBOARDS

The blackboard problem solving methodology originated approximately 10 years ago and has been evolving ever since. The hallmarks of a blackboard system are:

- A global data store holding input data and hypotheses about the solution of the problem derived from that data. Related information is kept together. This data store is known as the blackboard.
- A collection of procedures for deriving hypotheses about the solution of the problem from the input data and/or from other hypotheses. Each procedure is specialized to operate on a particular portion of the blackboard. These procedures are known as knowledge sources.
- A mechanism for invoking a knowledge source on relevant parts of the blackboard. A knowledge source is invoked on a particular piece of the blackboard when the invocation would incrementally advance the solution of the problem. This mechanism is known as the control structure.

Each of these hallmarks is described abstractly in the remainder of this section with simple examples appearing in the next

The blackboard holds the state of the problem solving system as the solution evolves. In conventional terms, the dimensionality of the state varies with time. The elements may be discretely or continuously valued. And the elements change values at discrete times. But such observations miss the most significant feature of the blackboard. It structures the information it holds.

Closely related input data or hypotheses are collected together in the form of blackboard nodes having certain attributes and values for those attributes. Related nodes form blackboard levels. All the nodes in a given level having the same attributes but (potentially) different attribute values. Levels can in turn form hierarchies of analysis or abstraction, usually with input data nodes at the base of each hierarchy. The most common nodal attributes are links between nodes on different levels. Such links connect hypotheses to input data or other hypotheses which support them. They can be links up and down levels within a hierarchy or they can be across hierarchies.

Knowledge sources transform the state of the problem solving system by adding nodes to the blackboard, by removing them, or by modifying their attribute values. Knowledge sources are effectively parametric procedures for transforming the state. A knowledge source could be invoked on any node at a given level or a tuple of nodes at one or more levels. It operates only on the node(s) upon which it is invoked plus those nodes linked directly or indirectly to them. Knowledge sources are also effectively typed procedures; a knowledge source can be invoked only on a node of a particular level or on a tuple of nodes, each of a particular level This feature of knowledge sources provides them with a degree of modularity. In particular, knowledge sources do not interact directly.

The procedure carried out by a knowledge source expresses knowledge of how to advance the problem solution. It is expressed in the creation, modification, and/or elimination of particular sorts of hypotheses in the form of nodes of particular levels. In this sense, a knowledge source is a specialist in the solution of some part of the overall problem. The details of the procedure can be expressed in any form. A typical form is a set of production rules and a policy for using them.

Each production rule specifies a logical condition on the attribute values of the node(s) upon which the knowledge source is invoked and an action to be carried out if that condition is true. Both the condition and action can be compound. The value of a compound condition is TRUE if the values of all its component conditions have TRUE values. A compound action is simply a sequence of individual nodal creations, deletions, or modifications. Evaluating a logical condition or modifying a node may require the application of complex numeric functions to attribute values. In this way, production rules mix symbolic and numeric computations.

Different policies for using a set of production rules allow at most one action to occur, or

multiple actions but never the same one twice, or the same one repeatedly. In the first case, the rules are scanned in order of definition with the scan terminating immediately if a rule's action is carried out. In the second case, the logical conditions of the rules are all tested before any actions take place. Then any actions are carried out in parallel. The third case is simply the second case repeated until no logical condition is TRUE. While this style of programming many seem bizarre at first, it has proved quite successful in past and existing blackboard systems.

A knowledge source describes the procedure by which it changes the blackboard when invoked. It also describes when it is invocable. The most general form of this description is a (possibly compound) logical condition on attribute values of the node(s) upon which it could be invoked. In this manner, a knowledge source resembles a production rule. The condition is parametric in the same sense that each knowledge source is parametric. As a result, the same knowledge source may be invocable on several nodes or tuples of nodes simultaneously. Each such combination of a knowledge source and a node or tuple of nodes is called a potential invocation. At any time, there are typically many potential invocations. The control structure determines the set of potential invocations, picks one, and causes it to be carried out.

Many blackboard systems do not use the most general form to describe when a knowledge source is invocable. They use events and logical combinations thereof. An event is a summary of a blackboard change. A knowledge source posts the appropriate event or events when it completes. A pointer to the affected node is associated with each event. These systems may also use events for an additional purpose as explained below.

The control structure is intended to operate in an opportunistic manner analogous to the manner in which people solve jigsaw puzzles. Initially, the puzzle solver scans for pieces with singular small-scale characteristics. If two such pieces have similar characteristics, they are tested for fit. Gradually, clusters of pieces accrete as the puzzle solver continues to scan through the unused pieces. Once the clusters become sufficiently large, scanning the pieces is replaced by searches for specific pieces to extend a cluster. But pieces plausibly belonging another cluster are tested for fit there if they are chanced upon during a search. Eventually, large clusters are recognized as connected on the basis of large scale characteristics and are jointed. If progress while searching for specific pieces bogs down, the puzzle solver reverts to scanning for pieces with similar characteristics for a time. It choses that activity which, at the moment, seems likely to make the best contribution to the overall solution of the problem.

A variety of techniques are used by the control structures of different blackboard systems to decide which potential invocation would, if carried out, make the best contribution to the overall solution. The topic is being actively researched. One system has an additional blackboard for handling hypotheses about the best choice (3) and another allows all potential invocations to be carried out in parallel (6).

Several blackboard systems use events in their control structures. After a particular event or sequence of events, particular knowledge sources are preferred to others. And they are prefered for invocation on the affected node or nodes. These same systems also use events to describe when a knowledge source is invocable. So the control structures of these systems need only attend to events and not to the blackboard nodes themselves.

Some of these blackboard systems also use expectations in their control structures. Expectations are posted by knowledge sources just as events are posted. Generally speaking, they are instructions to invoke a particular knowledge source on a particular node or nodes when, if ever, a certain event or pattern of events occurs involving the node(s). Expectations can also be negative. Such expectations cause a particular knowledge source to be invoked if a certain event or pattern of events does not occur within a specified time interval.

3. BLACKBOARDS ILLUSTRATED

Consider the problem of producing a situation map of aircraft flying over an area of interest. The situation map is based on track reports from an air surveillance radar tracking system, emitter/bearing reports from an ELINT system sensing airborne radar emissions, and warnings from a human intelligence system. The warnings are that particular aircraft or groups

of aircraft may soon enter the area of interest with particular objectives in mind. The situation map should identify the type of each aircraft as well as its current position and velocity. The radar track reports are regular for aircraft in the area of interest. The ELINT reports are intermittent by comparison. There are no reports unless an emitter is on. And the detection range of an active emitter can depend on its type and, in some cases, on the aircraft's aspect. ELINT reports are also less accurate geometrically than radar reports. Intelligence reports are generally less frequent than the ELINT reports, but can be updated rapidly on occasion.

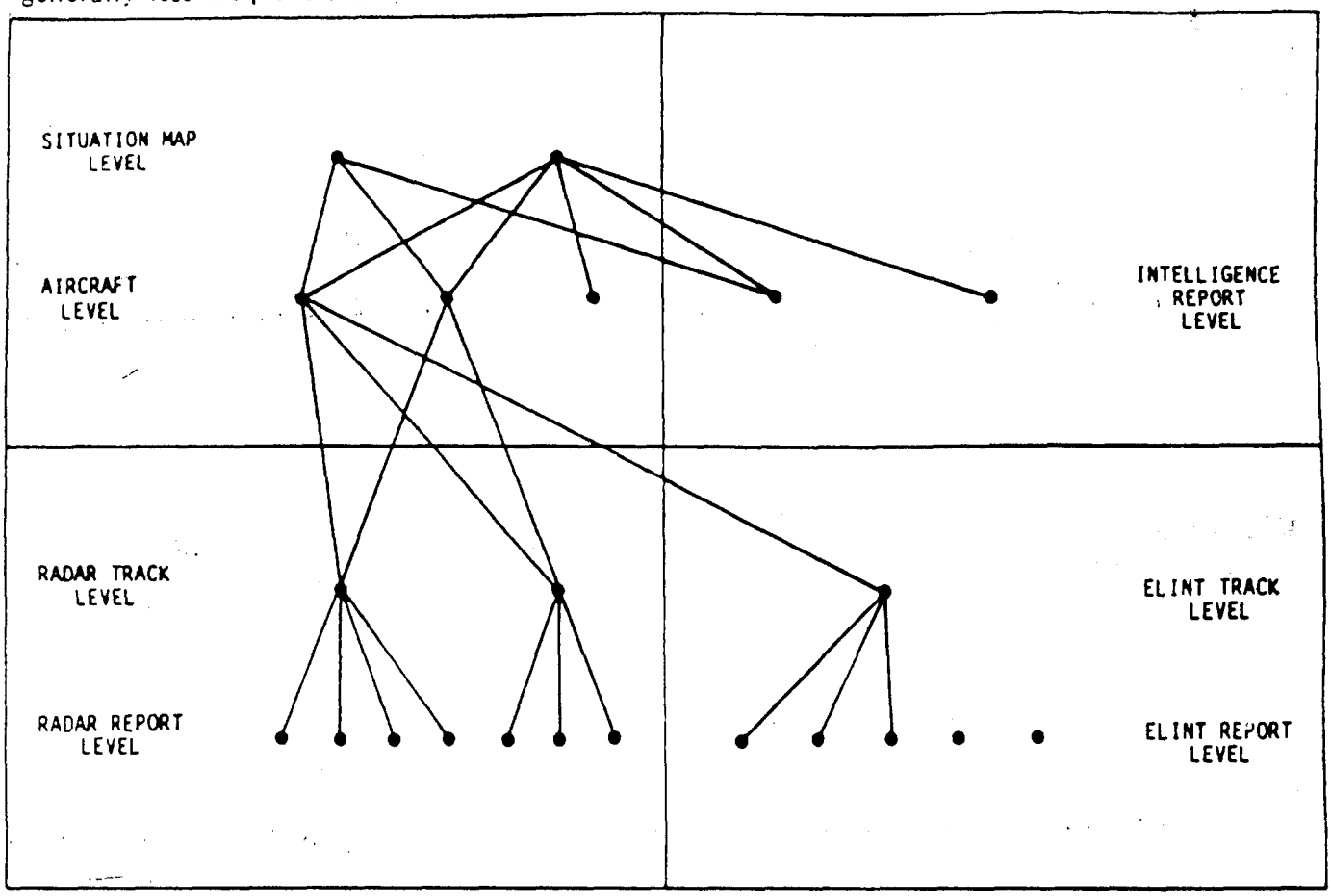


Figure 3-1: A Blackboard with 7 levels of nodes in 4 hierarchies

Figure 3-1 illustrates a possible blackboard configuration during the course of solving this problem. There are seven levels on the blackboard, a typical number. The situation map and aircraft levels form one hierarchy of levels. Nodes on these two levels hierarchically express alternative hypotheses about the map of aircraft in the area of interest. Two situation map hypotheses exist in this case, both including the same two hypothetical aircraft and one including a hypothetical third aircraft as shown by links between the corresponding nodes in the figure. One attribute of a situation map node is thus a set of component aircraft nodes. Hypothesis credibility is also a situation map node attribute. *A posteriori* probability would be a reasonable credibility measure. The value of that attribute is a function of the credibilities of the supporting aircraft hypotheses.

The intelligence report level is treated as a separate, degenerate hierarchy in the figure. The figure shows two intelligence report nodes. Links indicate that one of these reports supports both situation map hypotheses while the second report supports only one of them. The credibility attribute value of each situation map node is also a function of the credibility of each intelligence report node linked to it.

The radar track and radar report levels form another hierarchy. So do the ELINT track and ELINT report levels. A sequence of report nodes is linked to a corresponding track node to represent the hypothesis that they were all caused by the same object, aircraft or emitter.

Similarly, the links between the aircraft nodes and both kinds of track nodes represent the hypothesis that the tracks are all of the same aircraft. The credibility of an aircraft hypothesis is a function of the credibilities of the two kinds of track hypotheses supporting it.

It will prove useful later to have explicit definitions of certain attributes of radar report and radar track nodes. We do so in pseudo-computerese as follows:

```
Level: radar-report
Attributes: report-time
            track-identifier
            state-estimate
                North position
                East position
                North velocity
                East velocity
            state-covariance
                ...
            associated-tracks

Level: radar-track
Attributes: last-associated-report
            report-history
            track-credibility
```

The names of the attributes suggest their intended meanings. But attributes are given pragmatic meaning by the way the attributes are manipulated by knowledge sources. They are analogous to the elements of a state vector in this sense.

Knowledge sources embody knowledge about how to solve a problem. Consider the following fragment of knowledge about radar tracking:

> A sequence of radar reports caused by a particular aircraft usually have the same track identifier. An exception may occur if two aircraft approach closely at some time, in which case the track identifiers are swapped at roughly the time of closest approach.

It can be converted into the following fragments of knowledge about collecting radar reports into radar tracks:

> Given a radar report node that is not associated with any radar track node and given a radar track node, if the radar report node's track identifier is the same as that of the radar track node's last associated radar report node, then associate them.
>
> Given two radar track nodes, if their histories of associated radar report nodes indicate a close approach, then create two new radar track nodes with histories composed by splitting the original track nodes' histories at the time of closest approach and rejoining them with the track identifiers swapped after that time.

A knowledge source based on the first of these fragments is expressed in pseudo-computerese as follows:

```
Applies-to:
        a-radar-track , a-radar-report

Invocation-condition:
        associated-tracks of a-radar-report =
                empty-set

Use-policy:
        all-true-once

Production-rule 1:
Condition:
        track-identifier of last-associated-report
```

```
                    of a-radar-track =
                track-identifier of a-radar-report

Action:
        last-associated-report of a-radar-track
                := link to a-radar-report ;
        report-history of a-radar-track
                :== link to a-radar-report ;
        associated-tracks of a-track-report
                := link to a-radar-track
```

Here ":=" symbolizes assignment, ":==" signifies addition to a set, and ";" sequences simple actions in a compound one.

The knowledge source is quite simple, with just one production rule. That is atypical. Knowledge sources using production rules typically employ between ten and thirty production rules. A knowledge source realizing the second fragment would be more complex. It would include one or more production rules used to determine whether a possible close approach occurred and when.

The details of any particular control structure are complex. And the motivation for that complexity is not apparent in an example involving just one or two knowledge sources and a few nodes. So no attempt is made to include control structure details in this illustration. A sketch of the blackboard changes one would prefer under particular circumstances provides a better feel for the control structure's gross behavior. It also illustrates how the different components of a blackboard system can come together to solve a problem.

Assume that no reports have been received of any sort by the blackboard system. Then one situation map node exists with no links to aircraft nodes. This represents the hypothesis that no aircraft are in the area of interest. Then an intelligence report is posted on the blackboard. It warns that some number of aircraft of a particular type or types are expected to enter the area during a specified time interval across a specified portion of the area's boundary. Aircraft nodes are then created with the appropriate types, all linked to a new situation map node. The credibility of this new situation map node is the same as that of the intelligence report. The credibility of the old situation map node is appropriately adjusted downward.

The radar track attribute of each new aircraft node is not filled in at this point. There are no radar track nodes yet. But an expectation is established that later examines newly created radar track nodes. If one is created in the appropriate time interval and the appropriate place, a link to that radar track becomes the value of the associated track attribute. If the expectation goes unsatisfied, the aircraft node is deleted and the credibility of each associated situation map is reduced. Whenever the credibility of a situation map node slips below a certain level, that node is also deleted. Any aircraft nodes linked only to that situation map node are also deleted. The credibilities of all remaining situation maps are then re-normalized.

Receipt of the first few radar track reports causes them to be posted on the blackboard, but no more. Only when three report nodes having the same track identifier appear on the blackboard is a radar track node created to represent the hypothesis that they are from a single aircraft. In this manner, the creation of false radar track nodes based on radar false alarms is largely avoided. The resulting node may then be linked to an existing aircraft node by the aforementioned expectation.

Failing that, a new aircraft node is created to which the new radar track node is linked. Then the cross-product is formed of the old situation map hypotheses and the pair of hypotheses that the radar track was or was not caused by an aircraft. One new situation map node is created corresponding to each existing one. The new situation map nodes are copies of the old nodes, each with a link to this aircraft node added. Some portion of the credibility of each old situation map hypothesis must also be transferred to the corresponding new hypothesis. At this point, the knowledge source which removes insufficiently credible situation map nodes is again applied to reduce the number of situation map hypotheses maintained.

The accretion of ELINT reports into ELINT tracks is similar to that of radar reports into radar tracks. But the creation an of ELINT track does not satisfy any expectations or trigger

the creation of an aircraft node. Rather it triggers a search for aircraft nodes of a type which could produce the sensed emission and which has a history of estimated positions (implicit in the radar tracks' report history) consistent with the ELINT track's history of bearings (similarly implicit). The ELINT track node is linked with any and all such aircraft nodes. The credibility of any such aircraft nodes is increased appropriately to reflect evidence that the hypothesis it represents is correct. Such a credibility increase must also be propagated up to the situation map nodes. Creation of a new aircraft node triggers a similar search for supporting ELINT tracks.

Prioritization among the knowledge sources carrying out the aforementioned actions can be relatively simple. The arrival of a new input datum should trigger a locus of activity on the blackboard which propagates up the network of levels, with pauses to spread down along different hierarchies as appropriate. All of the activity directly triggered by one datum should be completed before the next input datum is posted. To keep the amount of inter-input processing reasonable, the diversity of hypotheses created in the normal course of processing must be limited. Thus as additional radar reports arrive, the posted nodes are simply associated with radar tracks on the basis of track identifiers as in the above knowledge source example. It would be possible to create track nodes expressing all possible hypothetical combination of track reports without regard to track identifiers. But the processing required to create, qualify, and eventually delete most of these nodes would be wasteful given the number of possible combinations.

But when should the control structure invoke the knowledge source which tests for a close approach of two aircraft and creates new track nodes to reflect a possible confusion of track identifiers? One answer would be after the completion of every invocation of the knowledge source associating a new radar report with an existing radar track. But that would mean frequent invocations, usually producing no change. An alternative is to invoke that knowledge source only when some other, less frequent, occurrence suggests the possibility of a close approach by two aircraft and consequent track identifier confusion be considered.

In the scheme described above, ELINT tracks are associated with an aircraft if they are consistent with the aircraft's hypothesized type and with the radar track. If the tracks are geometrically consistent but the nature of the tracked emission is inconsistent with the aircraft type, one possibility is that the aircraft hypothesis was wrong with regard to type and should be discarded or modified. But another possibility is that the radar track history actually corresponds to two different aircraft at two different times due to a track identifier confusion during a close approach. If ELINT tracks are already linked with the aircraft node as support for the hypotheses, the possibility of a close approach should be investigated first.

The above sketch does not reflect the only manner in which the example problem might be solved. It reflects various options for incrementally advancing the problem solution. Choosing which option to use in a particular situation can require subtlety if one wishes to be computationally efficient. Not illustrated are the additional subtleties of advising the control structure how to achieve that sequencing. Experience is required to make such choices wisely. Experience is also important in the construction of knowledge sources, the choice of blackboard levels, and the selection of nodal attributes. Simple examples can only suggest the subtleties involved.

4. SUITABILITY OF BLACKBOARDS

The above sketch of possible blackboard changes illustrates a major reason why the blackboard problem solving methodology is suitable for multi-system report integration. The ordering of changes adapts appropriately to the arrival of very different sorts of input data in different orders.

If any intelligence report involving a particular aircraft arrives after radar track reports corresponding to it, the hypothesis that it exists will still have been formed. The credibility of the situation map hypotheses supported by that aircraft hypothesis will be increased once the intelligence report is incorporated into the support for those situation map hypotheses. ELINT reports are not discarded immediately if they do not confirm an existing aircraft hypothesis. They are saved for possible confirmation in the future. And exceptional occurrences need be

considered only when evidence suggests they occur. The close approach of two aircraft leading to track identifier confusion being the case in point.

This adaptability in the operation of a blackboard system is a consequence of the control structure's opportunistic invocation of knowledge sources, the knowledge sources' modularity of forming or altering hypotheses, and the blackboard's structured composition of hypotheses. Any knowledge source can be invoked after any other completes, depending on the state of the blackboard, i.e., of the problem's solution, at that point in time.

The blackboard methodology also provides a means for managing the complexity of large multi-system report integration problems. Knowledge sources are modular in their applicability to all nodes of a given level, or tuples of given levels, but only to those nodes. Modularity is also achieved by expressing a partial problem solution as hypotheses supported by a hierarchy, or a set of linked hierarchies, of sub-hypotheses ultimately based on input data. Solution to individual parts of a particular multi-system report integration problem can be conceptualized and implemented without dwelling on the details of how the results of solving one part are used in the solutions of other parts.

Standard algorithms can be used where appropriate to solving part of the problem. But special pre- or post-processing may be required. Such pragmatic features of a standard algorithm's use in a particular context can be isolated from the algorithm itself by encapsulating them in separate knowledge sources. Explicitly separating formal and heuristic aspects of a problem's solution can highlight the heuristic aspects. It illuminates the assumptions, explicit or implicit, upon which they are based. Modifying the heuristic aspects without compromising the formal aspects also becomes easier.

5. WORK IN PROGRESS

The Heuristic Programming Project Group of Stanford's Knowledge System Laboratory is trying to

- realize a new generation of software architectures using parallel computation to speed up AI applications and
- specify multiprocessor system architectures for carrying out those computations efficiently.

Among the issues being investigated are

- recognition of opportunities for parallelism in the solution to a problem and
- expression of that potential parallelism in a problem solving framework that can exploit it.

In particular, this effort is focusing on signal understanding problems and blackboard-like frameworks.

Blackboard systems appear to be intrinsically parallel. At any time, there can be many potential invocations of knowledge sources. Those involving different nodes seem eligible for parallel execution. Within knowledge sources, production rule conditions could be evaluated in parallel. And some production rule actions could be safely executed in parallel. Currently two different blackboard systems are under development, each investigating a different approach to expressing opportunities for parallel computation or requirements for serial computation. Applications of these experimental systems used in evaluating their effectiveness.

The focus on signal understanding problems follows in large part from the focus on blackboard systems. The two mate well. But signal understanding problems are important in their own right. When signal understanding is defined broadly, it includes sensor data fusion and multi-system report integration. That class of problems is large and of considerable interest to the military.

Two signal understanding problems have been investigated so far as part of the current

project. They are referred to as the TRICERO/ELINT and AIRTRAC problems. While generally similar, each problem is expected to push the research into recognizing opportunities for, and expressing, parallel computation in different directions.

In the TRICERO/ELINT problem, streams of ELINT emitter/bearing measurements must be combined to estimate the flight paths and operating modes of non-cooperating aircraft. The problem is named after ESL's TRICERO blackboard system for solving a problem of which this one is just a component. The knowledge of how to solve the TRICERO/ELINT problem has already been worked out, albeit without attention to opportunities for parallel computation. So work on this problem is further along.

The AIRTRAC problem is recognizing aircraft flying across a national border and heading for particular airfields used by smugglers. The smugglers' aircraft must be picked out of the normal air traffic across that border. To solve the problem, aircraft destinations must be recognized, not just flight paths and types. Streams of radar reports from multiple radar systems are available. But the low altitude coverage of those radars is assumed to be limited and the smugglers are assumed to know the coverage limits. So smugglers can try to avoid detection. They can also maneuver their aircraft evasively to disrupt tracking. Such behavior is a sure sign of a smuggler's aircraft, but makes the recognition of a destination difficult.

To complicate the AIRTRAC problem further, distributed aeroacoustic tracking systems using modest batteries of acoustic sensor arrays(1,7) are placed across large holes in radar coverage. These systems provide tracking reports within their limited coverage. Because such systems are passive and readily moved, the smugglers are assumed to be unaware of their coverage and so unable to avoid detection by these systems. These systems also use acoustic signature information to provide aircraft class estimates along with tracking reports.

Initial solutions to both problems should be completed in both experimental blackboard systems by the end of the year. Moreover, each solution should have been applied to several problem scenarios on realistic simulated multiprocessors. These experiments will determine how much parallelism was realized and may suggest alternative ways of realizing more parallelism.

6. REFERENCES

(1) J.R. Delaney and R.R. Tenney, "Broadcast Communication Policies for Distributed Aeroacoustic Tracking", Proceedings of the 8th MIT/ONR Workshop on C^3 Systems, Cambridge, MA, July, 1985, pp.195-199.

(2) L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy, "The HEARSAY-II Speech Understanding System: Integrating Knowledge To Resolve Uncertainty", Computing Surveys, v. 12, December 1980, pp. 213-253. Also reprinted in (8).

(3) B.Hayes-Roth, "A Blackboard Architecture for Control", Artificial Intelligence, vol. 26, no. 3, July 1985, pp. 251-321.

(4) H.P. Nii and E.A. Feigenbaum, "Rule-Based Understanding of Signals", in D.A. Waterman and F. Hayes-Roth, Pattern-Directed Inference Systems, Academic Press, San Francisco, 1978, pp. 483-501.

(5) H.P. Nii, E.A. Feigenbaum, J.J. Anton, and A.J. Rockmore, "Signal-to-Symbol Transformation: HASP/SIAP Case Study", AI Magazine, vol 3, no. 2, Spring 1982, pp 23-35.

(6) J. Rice, "POLIGON: A System for Parallel Problem Solving", Knowledge Systems Laboratory Technical Report 86-19, Stanford University, 1986

(7) R.R. Tenney and J.R. Delaney, "A Distributed Aeroacoustic Tracking Algorithm", Proceedings of the 1984 American Control Conference, San Diego, CA, June 1984, pp 1440-1450.

(8) B.L. Webber and N.J. Nilsson (eds.), Readings In Artificial Intelligence, Tioga Press Company, Palo Alto, 1981.

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.